GeoSteiner 5.2

User's Guide and Reference Manual

Contents

Preface

This manual documents GeoSteiner version 5.2 — an optimization software package for solving Steiner tree problems. GeoSteiner version 4.0 was a proprietary commercial product, that was released in substantially identical form under an open source form beginning with version 5.0. GeoSteiner version 3.1 is still available from `www.geosteiner.com` under an academic license, but is no longer supported.

Version 5.2 contains significant improvements over the previous version (GeoSteiner 3.1); these improvements are both functional and structural. By far the largest structural change is that the core optimization algorithms have now been encapsulated into a library of callable subroutines. This greatly facilitates the incorporation of GeoSteiner into other applications. Indeed, the old familiar standalone programs from version 3.1 have now been completely re-implemented to use only the documented library interfaces. The ability to use function calls instead of program calls from applications provides for more efficient solution of (large) series of problem instances; such applications occur frequently in, e.g., VLSI layout. In addition, the library interfaces provide greater control of the solution process when needed.

The authors would like to thank Benny K. Nielsen, who has been the programmer on the callable library project. In addition, Benny has written a major part of the new FST generator for uniformly-oriented Steiner trees.

Copenhagen/Washington, January 2017

David M. Warme
Pawel Winter
Martin Zachariasen

1 Introduction

GeoSteiner is a software package for solving Steiner tree problems. The package currently solves the following NP-hard problems in the plane[1]:

- Euclidean Steiner tree problem.
- Rectilinear Steiner tree problem.
- Uniformly-oriented Steiner tree problem (including hexagonal and octilinear Steiner tree problems).

In addition, the package gives the user access to a powerful solver for

- Minimum spanning tree in hypergraph (MSTHG) problem.

The solver for this NP-hard problem is used as a subroutine in the solution of the above geometric problems.

GeoSteiner is written in ANSI C. The code makes heavy use of linear programming (LP); the public domain LP-solver `lp_solve` is included (in a significantly modified form). However, the package also supports `CPLEX`, a proprietary product of IBM Inc., which is perhaps the fastest and most robust LP-solver available. The authors of GeoSteiner strongly recommend that you use CPLEX if at all possible — especially for production applications or published computational studies. The core callable library requires no supplementary software or libraries (except the CPLEX library if GeoSteiner is configured to use CPLEX as its LP solver).

In this introductory section we first define the problems that are solved by GeoSteiner, and give some fundamental definitions and acronyms used throughout this manual (Section 1.1). Then an introduction to the callable library and associated stand-alone problems is given (Sections 1.2 and 1.3). Finally, we give some historic background on GeoSteiner in Section 1.4.

[1]Problem definitions are given in Section 1.1.

1.1 Steiner tree problems

Given a metric and a (finite) set of points in the plane, also denoted terminals, a *Steiner minimum tree* (SMT) is a shortest possible interconnection of the terminals. This interconnection must be a tree, and may contain junctions that are not among the terminals, so-called *Steiner points*. In Figure 1 we show four different SMTs for the same set of terminals. These are SMTs under the Euclidean, rectilinear, hexagonal and octilinear metric, respectively.

All metrics currently handled by GeoSteiner are *uniformly oriented metrics*: Given a set of $\lambda \geq 2$ uniformly oriented directions in the plane, the distance between two points is defined to be the length of a shortest path in which all line segments have one of the given directions. As special cases we have the rectilinear ($\lambda = 2$), hexagonal ($\lambda = 3$), octilinear ($\lambda = 4$) and Euclidean ($\lambda = \infty$) metric.

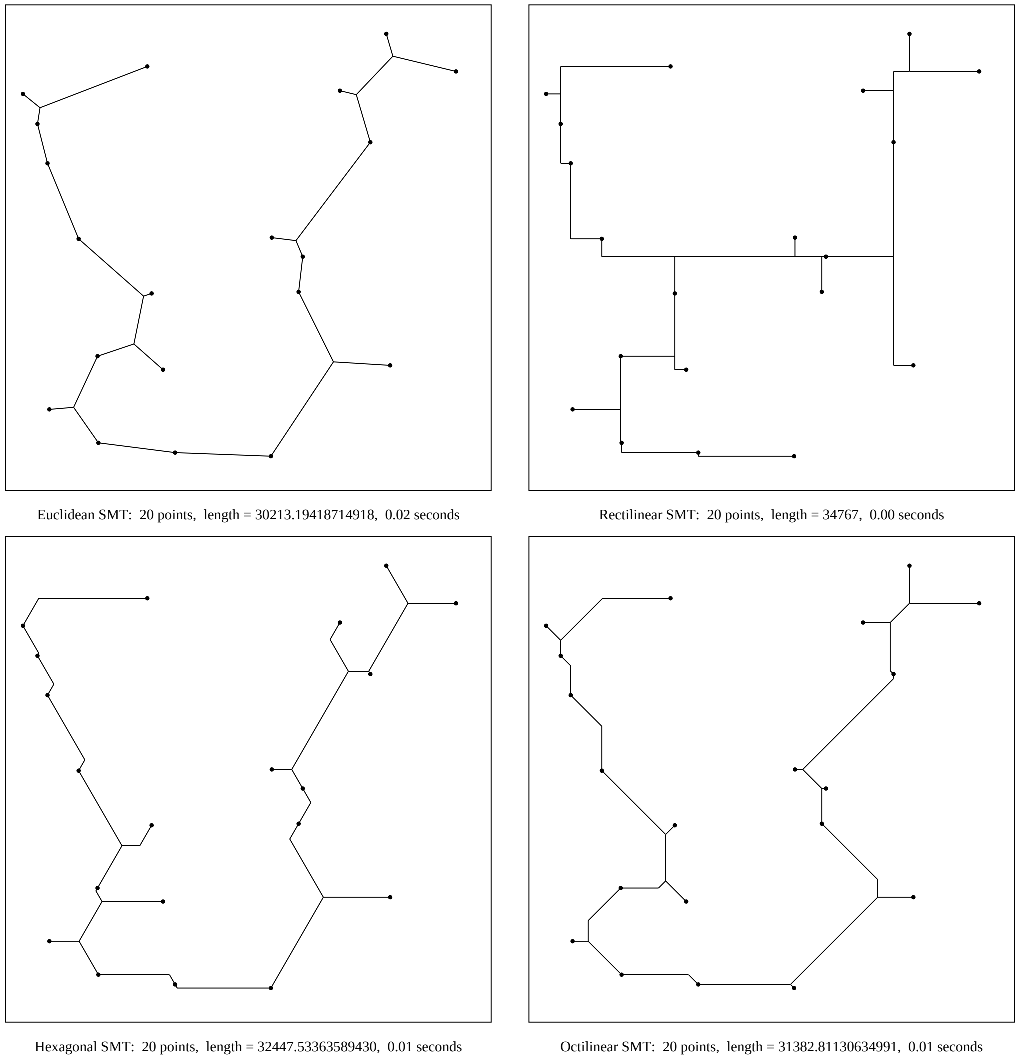
If we break an SMT at all terminals having two or more incident edges, each component will be a so-called *full Steiner tree* (FST). These are trees in which all terminals are leaves. The fact that the number of terminals in each FST in an SMT usually is small is what makes it possible to solve large problem instances to optimality. More specifically, the algorithms employed by GeoSteiner first generate a set of FSTs known to contain an SMT as a subset, and then the shortest possible union of FSTs interconnecting all terminals is selected; we say that the FSTs are concatenated. The concatenation problem is equivalent to finding a *minimum spanning tree in a hypergraph* (problem MSTHG). An efficient solver for this subproblem forms a cornerstone of GeoSteiner.

1.2 Callable library

The kernel of GeoSteiner is the callable library. Both high-level and low-level interfaces are provided. Also included are powerful routines for manipulating various algorithmic parameters, handling messages, and accessing problem instance data in various formats.

The high-level interfaces give the user easy access to the basic algorithms in the library. Problem instances are given as simple arrays, and the functions return optimal solutions to the problem instances.

Low-level interfaces are provided for users who need more control over the solu-



Euclidean SMT: 20 points, length = 30213.19418714918, 0.02 seconds

Rectilinear SMT: 20 points, length = 34767, 0.00 seconds

Hexagonal SMT: 20 points, length = 32447.53363589430, 0.01 seconds

Octilinear SMT: 20 points, length = 31382.81130634991, 0.01 seconds

Figure 1: SMTs for the same set of terminals under different metrics (output from GeoSteiner).

tion process. Also, the low-level interfaces are used by the stand-alone programs that accompany the callable library. For more details on the design and structure of the callable library, please consult the user's guide in Section 2 and the callable library reference manual in Section 3.

1.3 Stand-alone programs

The stand-alone programs are provided for users who would like to solve Steiner tree problems without writing their own application programs. For example, if the coordinates of the given problem instance are given in a file, the stand-alone programs give the user the opportunity to solve the instance and make a postscript plot of the solution. A complete list of all stand-alone programs, including documentation of their invocation options, and examples of their use are given in Section 4.

1.4 Historic note and literature

To the best knowledge of the authors, as of January 2017, GeoSteiner represents the computational state of the art for geometric Steiner tree problems in the plane under each of the following metrics:

- Euclidean
- Rectilinear
- Uniformly oriented metrics

Furthermore, GeoSteiner has held this dominant position continuously since at least 1998. During the 11th DIMACS Implementation Challenge (December, 2015) no other algorithms world-wide were entered into any of these problem categories. (Because GeoSteiner was only entrant in each of these categories, no competition was performed — which is why GeoSteiner does not appear in any of the official DIMACS 11 competition results.)

The "GeoSteiner" name was coined (and is therefore "owned") by Pawel Winter, whose seminal program GEOSTEINER started it all back in 1985 [7]. In 1996 Winter and Zachariasen published an improved algorithm called "GeoSteiner96" [8].

On the other hand, Warme's first Steiner tree code was the Salowe-Warme algorithm in 1993, which used backtrack search to concatenate rectilinear FSTs [3]. In 1998, Warme's Ph.D. dissertation [5] described a new branch-and-cut code for finding minimum spanning trees in arbitrary hypergraphs — which was applied to the FST concatenation problem for both rectilinear and Euclidean FSTs.

The first distribution of the combined code therefore represented the "third version" of each group's code, and it was thus named GeoSteiner version 3.0. This and subsequent versions continue that naming convention.

The algorithms in GeoSteiner 3.0 are based on those described in [6, 8, 9].

GeoSteiner 4.0 was a proprietary commercial product which introduced the callable library interfaces, and support for solving uniformly-oriented Steiner trees [2]. In addition, a number of minor improvements were made throughout the code.

Upon termination of commercial operations in 2015, the GeoSteiner code was released once again in open source form as GeoSteiner version 5.0, and its various successors.

2 Callable Library User's Guide

In this section we give a number of examples of using the callable library. We start with a few simple uses of the high-level functions, then move to the low-level interfaces, and finally, we discuss the use of callback functions.

2.1 High-level interfaces

Any application program that uses the GeoSteiner library must include the GeoSteiner header file `geosteiner.h`. Furthermore, the GeoSteiner environment must be opened using the function **gst_open_geosteiner()** as described in Section 3.4 on page 18.

Our first example, shown in Figure 2, computes an Euclidean SMT for the points $(0, 0)$, $(0, 1)$, $(1, 0)$ and $(1, 1)$. After having successfully opened the GeoSteiner environment, we use the high-level function **gst_esmt()** to compute the SMT (see page 30). As arguments we first pass the number of terminals, here 4, and then a double array **terms** that holds the terminal point coordinates. Then follow the variables **length**, **nsps** and **sps**, in which the length of the computed SMT, the number of Steiner points and the coordinates of the Steiner points will be returned. The remaining arguments to **gst_esmt()** are all given as NULL, causing corresponding inputs to assume default values, and corresponding outputs to be ignored; in particular the edges of the optimal solution and the solution status are all ignored, and default values are assumed for all GeoSteiner parameters.

In the program we assume that **gst_esmt()** returns successfully — the return code is not checked — and then we print the length of the SMT and the (two) Steiner points. Finally, we close the GeoSteiner environment and the program ends. We encourage you to run the `demo1` program that comes with the GeoSteiner distribution.

Our next example, `demo2`, computes a series of SMTs for randomly generated points sets (Figure 3). This program has two command line parameters: Firstly, the λ-value for the uniformly oriented metric that is to be used — where $\lambda = 0$ is defined to be the Euclidean metric; secondly, the required maximum excess from optimum in percent. As an example, "`demo2 4 1`" computes Steiner trees using the octilinear metric whose length are at most 1% from optimum.

```
#include "geosteiner.h"
#include "stdlib.h"

int main (int argc, char** argv)
{
        double terms [8] = { 0, 0,
                             0, 1,
                             1, 0,
                             1, 1 };

        int i, nsps;
        double length, sps [4];

        /* Open GeoSteiner environment */
        if (gst_open_geosteiner () != 0) {
                printf ("Could not open GeoSteiner.\n");
                exit (1);
        }

        /* Compute Euclidean Steiner tree */
        gst_esmt (4, terms, &length, &nsps, sps, NULL, NULL, NULL, NULL);

        /* Display information about solution */
        printf ("Steiner tree has length %f\n", length);

        for (i = 0; i < nsps; i++) {
                printf ("Steiner point: (%f, %f)\n", sps[2*i], sps[2*i+1]);
        }

        /* Close GeoSteiner environment */
        gst_close_geosteiner ();

        exit (0);
}
```

Figure 2: Demo program that computes an Euclidean SMT for four terminals (demo1.c)

In the program we first read the command line parameters and then create a metric object (see Section 3.7) that corresponds to the given command line parameter. Then we create a default parameter set (see Section 3.6) and change the `GST_PARAM_GAP_TARGET` parameter (see Appendix A). Finally, we compute the SMTs using the high-level function **gst_smt()** which takes the metric object and parameter set as arguments. Only the SMT length is returned from **gst_smt()**, and based on this the total length of all SMTs is computed and displayed.

Our third example, shown in Figure 4, is similar to the previous example, but instead of generating the terminal coordinates using a random number generator, we read the terminal coordinates from input. The input is assumed to be in the OR-library format[2]. The program reads every instance in the file and computes an SMT for each. The metric is given as the first command line parameter to the program. Furthermore, the maximum FST size (number of terminals) can be specified as the second command line parameter. By giving (small) bound on the FST size, the running time of FST generation may decrease significantly — at the cost of not necessarily returning the optimal solution.

2.2 Low-level interfaces

The low-level interfaces completely separate FST generation and FST concatenation, the two components of the exact algorithm used by GeoSteiner. Thus it is possible to use alternative FST generation or concatenation algorithms — or to store away generated FSTs and concatenate them at a later time.

Another advantage of using the low-level interfaces is the greater control they provide over the solution process, in particular with respect to solving the FST concatenation problem. For most large instances the FST concatenation problem — which is equivalent to solving a MSTHG problem — is by far the most time-consuming part of the solution process.

As for the high-level interfaces, programs that use the low-level interfaces must include the GeoSteiner header file and open the GeoSteiner environment. In the example given in Figure 5 we construct a large random terminal set, generate the rectilinear FSTs, and set up a solution state object for the MSTHG problem (see Section 3.11). One of the parameters passed to the solution state object is

[2]OR-library: http://www.ms.ic.ac.uk/info.html

```
#include <math.h>
#include <stdlib.h>
#include "geosteiner.h"

#define NUM_INSTANCES   10
#define NUM_TERMS       50

int main (int argc, char** argv)
{
        int i, j, lambda = 2;
        double terms[2*NUM_TERMS], length, total_length = 0.0, max_excess = 0.0;
        gst_metric_ptr metric;
        gst_param_ptr params;

        /* Read command line parameters (metric and max. excess in percent) */
        if (argc >= 2) lambda     = atoi (argv[1]);
        if (argc >= 3) max_excess = atof (argv[2]);

        /* Open GeoSteiner environment */
        if (gst_open_geosteiner () != 0) {
                printf ("Could not open GeoSteiner.\n");
                exit (1);
        }

        /* Set up metric */
        switch (lambda) {
        case 0: /* Euclidean metric */
                metric = gst_create_metric (GST_METRIC_L, 2, NULL); break;
        case 2: /* Rectilinear metric */
                metric = gst_create_metric (GST_METRIC_L, 1, NULL); break;
        default:/* General uniform metric */
                metric = gst_create_metric (GST_METRIC_UNIFORM, lambda, NULL);
        }

        /* Set up parameter set */
        params = gst_create_param (NULL);
        gst_set_dbl_param (params, GST_PARAM_GAP_TARGET, 1.0 + (max_excess/100.0));

        /* Generate NUM_INSTANCES random instances with NUM_TERMS terminals */
        srand48 (1);
        for (i = 1; i <= NUM_INSTANCES; i++) {

                /* Generate random points with coordinates in range 0..9999 */
                for (j = 0; j < 2*NUM_TERMS; j++)
                        terms[j] = floor (drand48() * 10000.0);

                /* Compute Steiner tree and print length */
                gst_smt (NUM_TERMS, terms, &length, NULL, NULL, NULL, NULL, NULL,
                         metric, params);
                printf ("Instance %2d has length %f\n", i, length);
                total_length += length;
        }
        printf ("\nTotal length of all instances is %f\n", total_length);

        /* Clean up */
        gst_free_metric (metric);
        gst_free_param (params);
        gst_close_geosteiner ();

        exit (0);
}
```

Figure 3: Demo program that computes SMTs for a series of randomly generated problem instances (demo2.c).

```
#include <stdlib.h>
#include "geosteiner.h"

int main (int argc, char** argv)
{
        int i, lambda = 2, num_instances, num_terms;
        double * terms, length, total_length = 0.0, max_fst_size = 0;
        gst_metric_ptr metric;
        gst_param_ptr params;

        /* Read command line parameters (metric and max. FST size) */
        if (argc >= 2) lambda       = atoi (argv [1]);
        if (argc >= 3) max_fst_size = atof (argv [2]);

        /* Open GeoSteiner environment */
        if (gst_open_geosteiner () != 0) {
                printf ("Could not open GeoSteiner.\n");
                exit (1);
        }

        /* Set up metric */
        switch (lambda) {
        case 0: /* Euclidean metric */
                metric = gst_create_metric (GST_METRIC_L, 2, NULL); break;
        case 2: /* Rectilinear metric */
                metric = gst_create_metric (GST_METRIC_L, 1, NULL); break;
        default:/* General uniform metric */
                metric = gst_create_metric (GST_METRIC_UNIFORM, lambda, NULL);
        }

        /* Set up parameter set */
        params = gst_create_param (NULL);
        if (max_fst_size >= 2)
                gst_set_int_param (params, GST_PARAM_MAX_FST_SIZE, max_fst_size);

        /* Read the number of instances and then the instances thenselves */
        scanf ("%d", &num_instances);
        for (i = 1; i <= num_instances; i++) {

                /* Read instance from stdin */
                scanf ("%d", &num_terms);
                terms = (double *) malloc (2*num_terms*sizeof(double));
                gst_get_points (stdin, num_terms, &terms, NULL);

                /* Compute Steiner tree */
                gst_smt (num_terms, terms, &length, NULL, NULL, NULL, NULL, NULL,
                         metric, params);

                printf ("Instance %5d has %5d terminals and length %f\n",
                        i, num_terms, length);
                total_length += length;
                free (terms);
        }
        printf ("\nTotal length of all instances is %f\n", total_length);

        /* Clean up */
        gst_free_metric (metric);
        gst_free_param (params);
        gst_close_geosteiner ();

        exit (0);
}
```

Figure 4: Demo program that computes SMTs for a series of instances read from an OR-library file (demo3.c).

`GST_PARAM_CPU_TIME_LIMIT`, which limits the amount of time spent in the solver before returning to the application program.

In the main loop of the program we run the MSTHG solver by calling **gst_hg_solve()**, passing the solution state object as an argument. When this function returns, we query the solution state object for the current solution status; this is done by calling **gst_get_solver_status()** which returns a code that represents the four possibilities (optimal solution, feasible solution, infeasible problem, no feasible solution yet).

If a feasible solution has been found, the current upper and lower bound is obtained by querying the solution state property list. In our example we repeat the main loop until we have found a feasible solution that is within the maximum specified excess from optimum.

2.3 Algorithmic callback functions

Although the implementation of the callback interface is rather incomplete at this time, callback functions provide the lowest-level — and perhaps most powerful — of all the interfaces in the GeoSteiner callable library. Callbacks are user-written functions. Such functions become callbacks by passing their address to suitable GeoSteiner routines. Once a function is established as a callback in this manner, GeoSteiner automatically invokes the function at the corresponding critical points in the branch-and-cut algorithm. For example, the user can provide callback routines that are invoked every time

- an LP is solved during the optimize / separate loop,
- processing of a branch-and-bound node completes,
- a new lower bound is obtained,
- a new upper bound is obtained,
- a branch variable is selected.

Callback functions permit the user to extend the GeoSteiner optimization algorithms by incorporating application specific knowledge into some of GeoSteiner's most critical decisions. As an example, the `bb` program (see Section 4) uses

```
#include <math.h>
#include <stdlib.h>
#include "geosteiner.h"

#define NUM_TERMS       1000
#define TIME_INTERVAL   2
#define MAX_EXCESS      0.1

int main (int argc, char** argv)
{
        int j, status, soln_status;
        double terms[2*NUM_TERMS], lb, ub, cpu;
        gst_hg_ptr hg;  gst_solver_ptr solver;  gst_param_ptr params;

        /* Open GeoSteiner environment */
        if (gst_open_geosteiner () != 0) {
                printf ("Could not open GeoSteiner.\n");
                exit (1);
        }
        /* Generate random terminals with coordinates in range 0..9999 */
        srand48 (1);
        for (j = 0; j < 2*NUM_TERMS; j++)
                terms[j] = floor (drand48 () * 10000.0);
        /* Generate full Steiner trees (default parameters) */
        hg = gst_generate_rfsts (NUM_TERMS, terms, NULL, &status);
        /* Set up solver and its parameters */
        params = gst_create_param (&status);
        gst_set_dbl_param (params, GST_PARAM_CPU_TIME_LIMIT, TIME_INTERVAL);
        solver = gst_create_solver (hg, params, &status);

        for (;;) {
                gst_hg_solve (solver, NULL);
                gst_get_solver_status (solver, &soln_status);
                switch (soln_status) {
                        case GST_STATUS_OPTIMAL:
                        case GST_STATUS_FEASIBLE:
                                gst_get_dbl_property (gst_get_solver_properties (solver),
                                                      GST_PROP_SOLVER_LOWER_BOUND, &lb);
                                gst_get_dbl_property (gst_get_solver_properties (solver),
                                                      GST_PROP_SOLVER_CPU_TIME, &cpu);
                                gst_hg_solution (solver, NULL, NULL, &ub, 0);

                                printf ("Time: %.2f. LB = %f, UB = %f, ratio = %f\n",
                                         cpu, lb, ub, ub/lb); break;
                        case GST_STATUS_INFEASIBLE:
                                printf ("Problem is infeasible!\n"); break;
                        case GST_STATUS_NO_FEASIBLE:
                                gst_get_dbl_property (gst_get_solver_properties (solver),
                                                      GST_PROP_SOLVER_CPU_TIME, &cpu);
                                printf ("Time: %.2f. No feasible solution found yet.\n", cpu);
                }
                if (soln_status == GST_STATUS_OPTIMAL) break;
                if ((soln_status == GST_STATUS_FEASIBLE) &&
                    (ub/lb < 1.0 + (MAX_EXCESS / 100.0))) break;
        }
        /* Clean up */
        gst_free_solver (solver);  gst_free_hg (hg);
        gst_free_param (params);   gst_close_geosteiner ();
        exit (0);
}
```

Figure 5: Demo program that computes a rectilinear Steiner tree (not necessarily minimal) for a large random terminal set. The upper bound/lower bound gap is displayed at fixed running time intervals (demo4.c).

callbacks to implement the `bb -r` switch: a callback function is defined that is invoked upon completion of every node. When invoked for the root node, the LP solution is fractional, and the `-r` switch was specified, this callback generates a postscript plot of the node's LP relaxation.

3 Callable Library Functions

3.1 Application programming interface

All declarations needed to use the GeoSteiner library in an application are defined in a single include file called `geosteiner.h`. All identifiers `#define`'d in the header file begin with the prefix "`GST_`". All structure or union tags and typedefs begin with the prefix "`gst_`". All functions provided by the library begin with the prefix "`gst_`".

All GeoSteiner library functions reside in a single library. On most systems the name will be `libgeosteiner.a` and linking is done with `-lgeosteiner`. A shared library is also possible on some systems. If GeoSteiner has been configured to use CPLEX as its LP-solver, then the CPLEX callable library must also be linked with the application program.

3.2 Design of library

The GeoSteiner library is designed to be completely re-entrant so that multiple problems can be solved serially or in a round-robin fashion. The current implementation might not yet completely satisfy this goal — especially concerning the various LP-solver interfaces and the way GeoSteiner interacts with them. We hope to eventually make the library fully thread-safe so that multiple problems can be solved in parallel within a single process address space on a multi-processor system. However, this ideal is not yet supported in the current version.

All output generated by the library (i.e., text that was written to stdout or stderr by previous versions of GeoSteiner) is now user-controllable. Various types of output have parameters that enable/disable their generation. This is achieved using so-called "channels" described in Section 3.13. By default, library routines are completely "quiet".

The library does not use any signals nor does it establish any signal handlers. These would be potential points of contention with applications that use the library. Instead, all asynchronous requests to alter or abort a GeoSteiner computation (e.g., to abort the solution process, force branching in lieu of constraint generation, etc.) are delivered by a single routine that is designed to be safe when called from a user-defined signal handler (see the description of the function **gst_deliver_signals()** on page 156 for more information).

3.3 Library objects

3.3.1 GeoSteiner environment

The GeoSteiner environment encapsulates licensing information and platform-specific data. If CPLEX is used as LP solver, the CPLEX environment is stored here.

The environment is a single *global* variable. No explicit user references to the environment are possible, but the environment *must* be initialized by calling the **gst_open_geosteiner()** function (see Section 3.4).

3.3.2 Parameter set

A parameter set holds values for all parameters used by the library. In order to change one or more parameters, the user creates a new parameter set and modifies the parameter(s) in this set. A pointer to the parameter set (type `gst_param_ptr`) is then passed to all functions for which these parameter settings should have effect. Whenever a GeoSteiner function accepts an argument of type `gst_param_ptr`, the user may pass a NULL pointer in which case the GeoSteiner library assumes default settings for all parameter values.

Parameter setting and querying functions are described in Section 3.6, while the individual parameters are described in Appendix A.

3.3.3 Problem instance

The problem instance object is a *hypergraph* that can be decorated with a variety of additional (and optional) data (see Section 3.9). By attaching information globally to the hypergraph, and to its vertices and edges, the problem to be solved becomes well-defined. In general we would like to construct a *tree* in the hypergraph. Problem instance objects have type `gst_hg_ptr`.

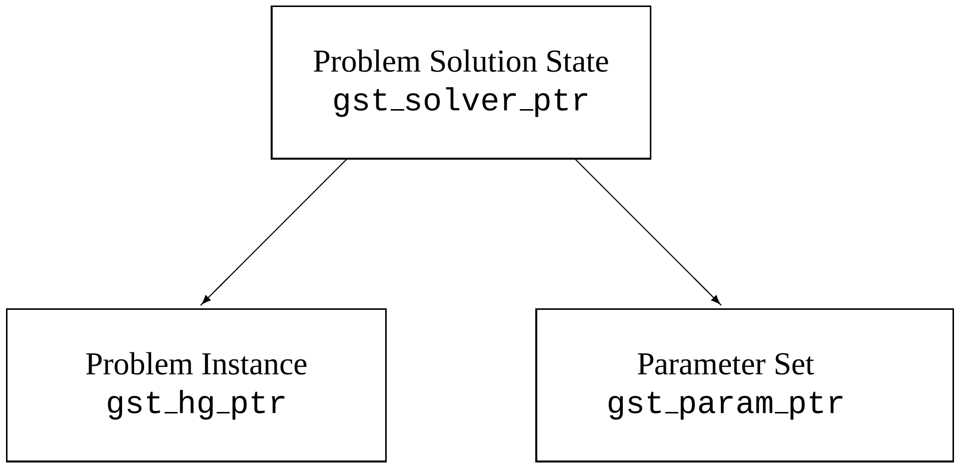


Figure 6: Problem solution state references a problem instance and a parameter set.

3.3.4 Problem solution state

The problem solution state object represents the "state" of some solution process for a given problem instance (see Section 3.11). The object can contain zero or more feasible (though not necessarily optimal) solutions to the problem. For a given problem instance, several problem solution state objects may be created. A problem solution state object refers to both a problem instance being solved and a parameter object (from which all necessary parameter values are obtained), as illustrated in Figure 6. The problem solution state object has type `gst_solver_ptr`.

3.3.5 Auxiliary objects

In addition to the four object classes described above, GeoSteiner uses objects for handling metrics, property lists, messages, and scaling information. A short introduction to these auxiliary objects is given in this section.

Metric A metric object identifies the method for computing distances between pairs of points. A metric object has type `gst_metric_ptr`, and can be passed as an argument to some of the functions in the callable library. For more information, see the examples given in Section 2.2, and the descriptions of the metric functions in Section 3.7.

Property List A property list contains auxiliary information about problem instances and solution state objects, e.g. the CPU time for FST generation (problem instance property) and the current lower bound in the MSTHG solver (solution state property). Property lists have type `gst_proplist_ptr`, and a property is known by its property identification number (see Section 3.8).

Channel All output messages from GeoSteiner are passed through user-controllable channels. A given channel may write its output to more than one output (screen/files). Channels have type `gst_channel_ptr` and are described in Section 3.13.

Scaling Information A set of points may have associated scaling information, that is, information about how the internal representation (double floating point values) should be scaled back to the original point coordinates. This is done in order to improve the numerical precision of GeoSteiner. Scaling information objects have type `gst_scale_info_ptr` and are described in Section 3.14.

3.4 Opening and closing GeoSteiner environment

The GeoSteiner environment encapsulates licensing information and platform-specific data. If CPLEX is used as LP solver, the CPLEX environment is stored in the GeoSteiner environment.

The environment is a single *global* variable. No explicit user references to the environment are possible, but the environment *must* be initialized by calling the **gst_open_geosteiner()** function before any other library functions can be invoked.

In the reminder of this section, we present each of the functions in the library related to the GeoSteiner environment.

gst_open_geosteiner

GeoSteiner can be in two major states *open* or *closed*. The initial state is always *closed*. This routine transitions GeoSteiner from the *closed* state to the *open* state by initializing the GeoSteiner environment. No other GeoSteiner library function may be called when GeoSteiner is *closed*. In a multi-threaded environment, it is the application's responsibility to ensure that no calls to other GeoSteiner library functions are either pending or initiated until GeoSteiner is in the *open* state — which begins as soon as this routine returns with a status code of zero.

Note that the function does *not* open the LP solver (e.g., CPLEX). This is done automatically the first time the LP solver environment is accessed; however, it can also be done explicitly using the **gst_open_lpsolver()** function. An existing CPLEX environment can also be attached to the GeoSteiner environment. See **gst_attach_cplex()**; this is only relevant for CPLEX versions of the library.

```
int gst_open_geosteiner (void);
```

Returns status code (which is zero if GeoSteiner was successfully opened).

Example:

```
if (gst_open_geosteiner()) {
   printf("GeoSteiner was not opened successfully.\n");
   exit(1);
}
```

gst_close_geosteiner

Transition GeoSteiner from the *open* to the *closed* state. Conceptually, GeoSteiner enters the *closed* state the very instant this routine is called. In a multi-threaded environment, it is the application's responsibility to ensure that no calls to other GeoSteiner library functions are pending at the time this routine is invoked.

```
int gst_close_geosteiner (void);
```

Returns error code (which is zero if GeoSteiner was successfully closed).

Example:

```
if (gst_close_geosteiner()) {
   printf("GeoSteiner was not closed successfully.\n");
   exit(1);
}
```

gst_version_string

Return GeoSteiner version number as a character string.

```
const char * gst_version_string (void);
```

Returns null-terminated string giving the GeoSteiner version number.

Example:

```
printf ("This is GeoSteiner version %s\n", gst_version_string());
```

gst_version

Return GeoSteiner version number as an integer with the following decimal interpretation: XXXYYYZZZ, where XXX is the major version, YYY is the minor version and ZZZ is the patch-level.

```
int gst_version (void);
```

Returns integer representing the version number.

Example:

```
int version = gst_version();
printf ("This is GeoSteiner version %d.%d.%d\n",
        (version / 1000000),
        (version / 1000) % 1000,
        (version % 1000));
```

gst_open_lpsolver

Initialize LP solver (e.g., CPLEX) environment. It is not necessary to open the LP solver explicitly, since this is done automatically the first time the LP solver is needed. However, it might be advantageous to ensure that the LP solver has been successfully opened and is available for use before starting a long run.

```
int gst_open_lpsolver (void);
```

Returns value zero if the LP solver was opened successfully or already was open.

Example:

```
if (gst_open_geosteiner()) {
   printf("GeoSteiner was not opened successfully.\n");
   exit(1);
}
if (gst_open_lpsolver()) {
   printf("LP solver was not initialized successfully.\n");
   exit(1);
}
/* At this point both GeoSteiner and the LP solver are opened... */
```

gst_close_lpsolver

Close LP solver environment. In the case where the LP solver was *attached*, e.g., using **gst_attach_cplex()**, then this routine detaches but does *not* close the LP solver.

```
int gst_close_lpsolver (void);
```

Returns value zero if the solver was closed successfully or already was closed.

Example:

```
if (gst_close_geosteiner()) {
   printf("LP solver could not be closed successfully.\n");
   exit(1);
}
```

gst_lpsolver_version_string

Return the name of the configured LP solver and its version number as a string.

```
const char* gst_lpsolver_version_string (void);
```

Returns zero-terminated string giving the LP solver name and version.

Example:

```
printf ("GeoSteiner used LP solver %s\n",
        gst_lpsolver_version_string());
```

gst_attach_cplex

Provided only for CPLEX versions of the library. Attach an existing CPLEX environment to GeoSteiner. Certain applications may wish to use CPLEX before, during and/or after they use GeoSteiner. This function permits such applications to use an existing CPLEX environment rather than letting GeoSteiner attempt to open CPLEX itself (which would fail if CPLEX were already open). A non-`NULL` CPLEX environment that was attached using **gst_attach_cplex()** will not be closed when **gst_close_geosteiner()** is called.

```
void gst_attach_cplex (struct cpxenv*  envp);
```

`envp`	CPLEX environment to be attached.

No return value.

Example:

```
/* Assume that envp is an existing CPLEX environment...*/

/* Open GeoSteiner */
if (gst_open_geosteiner()) {
   printf("GeoSteiner was not opened successfully.\n");
   exit(1);
}

/* Attach existing CPLEX environment */
gst_attach_cplex(envp);

/* Now envp is the CPLEX environment used by GeoSteiner... */

/* Detach CPLEX environment and close GeoSteiner */
gst_detach_cplex();
gst_close_geosteiner();
```

gst_detach_cplex

Provided only for CPLEX versions of the library. Detach and return a previously attached CPLEX environment. Does not close the CPLEX environment.

```
struct cpxenv* gst_detach_cplex ();
```

Return value is `NULL` if no CPLEX environment is currently attached.

An example is given with the documentation of **gst_attach_cplex()** on page 26.

3.5 High-level optimization functions

The high-level functions give the user easy access to the basic algorithms in the library. There are two types of functions: Firstly, there are functions that solve Steiner tree problems in the plane by passing a set of point coordinates; secondly, the MSTHG problem can be solved by giving a description of the hypergraph instance.

All functions have a parameter set as argument. This parameter set can be created and modified using the functions described in Section 3.6. However, default parameters are used for all parameters if a `NULL` pointer is passed as parameter set.

gst_smt

Given a set of points (or terminals) in the plane, construct an SMT for the points. The metric used for the SMT construction must be specified. (Dedicated functions for specific metrics are given on the following pages.) The length of the constructed SMT, the Steiner points and the list of line segments in the SMT are returned.

Any of the output parameters may be set to `NULL` if the corresponding output is not needed. It is the responsibility of the user to allocate sufficient memory for the output arrays.

```
int gst_smt (int            nterms,
             double*        terms,
             double*        length,
             int*           nsps,
             double*        sps,
             int*           nedges,
             int*           edges,
             int*           status,
             gst_metric_ptr metric,
             gst_param_ptr  param);
```

`nterms`	Number of points (or terminals).
`terms`	Input point coordinates $(x_1, y_1, x_2, y_2, \ldots)$.
`length`	Length of computed SMT.
`nsps`	Number of Steiner points.
`sps`	Steiner point coordinates.
`edges`	Edges of SMT (terminals have index 0 to `nterms`-1 while Steiner points have index `nterms` and up).
`status`	Solution status code (see page 114).
`metric`	Metric object (see Section 3.7).
`param`	Parameter set (`NULL`=default parameters).

Returns value zero if an SMT was computed and non-zero otherwise. See Figure 3 on page 9 or the example file `demo2.c` for an example of how to use **gst_smt()**.

gst_esmt

Given a set of points (or terminals) in the plane, construct an *Euclidean* SMT for the points. The length of the constructed SMT, the Steiner points and the list of line segments in the SMT are returned.

Any of the output parameters may be set to `NULL` if the corresponding output is not needed. It is the responsibility of the user to allocate sufficient memory for the output arrays.

```
int gst_esmt (int            nterms,
              double*        terms,
              double*        length,
              int*           nsps,
              double*        sps,
              int*           nedges,
              int*           edges,
              int*           status,
              gst_param_ptr  param);
```

`nterms`	Number of points (or terminals).
`terms`	Input point coordinates $(x_1, y_1, x_2, y_2, \ldots)$.
`length`	Length of computed SMT.
`nsps`	Number of Steiner points.
`sps`	Steiner point coordinates.
`edges`	Edges of SMT (terminals have index 0 to `nterms`-1 while Steiner points have index `nterms` and up).
`status`	Solution status code (see page 114).
`param`	Parameter set (`NULL`=default parameters).

Returns value zero if an SMT was computed and non-zero otherwise.

An example is given in Section 2.1.

gst_rsmt

Given a set of points (or terminals) in the plane, construct a *rectilinear* SMT for the points. The length of the constructed SMT, the Steiner points and the list of line segments in the SMT are returned.

Any of the output parameters may be set to NULL if the corresponding output is not needed. It is the responsibility of the user to allocate sufficient memory for the output arrays.

```
int gst_rsmt (int            nterms,
              double*        terms,
              double*        length,
              int*           nsps,
              double*        sps,
              int*           nedges,
              int*           edges,
              int*           status,
              gst_param_ptr  param);
```

nterms	Number of points (or terminals).
terms	Input point coordinates $(x_1, y_1, x_2, y_2, \ldots)$.
length	Length of computed SMT.
nsps	Number of Steiner points.
sps	Steiner point coordinates.
edges	Edges of SMT (terminals have index 0 to nterms-1 while Steiner points have index nterms and up).
status	Solution status code (see page 114).
param	Parameter set (NULL=default parameters).

Returns value zero if an SMT was computed and non-zero otherwise.

An example is given in Section 2.1.

gst_osmt

Given a set of points (or terminals) in the plane, construct an *octilinear* SMT for the points. The length of the constructed SMT, the Steiner points and the list of line segments in the SMT are returned.

Any of the output parameters may be set to `NULL` if the corresponding output is not needed. It is the responsibility of the user to allocate sufficient memory for the output arrays.

```
int gst_osmt (int            nterms,
              double*        terms,
              double*        length,
              int*           nsps,
              double*        sps,
              int*           nedges,
              int*           edges,
              int*           status,
              gst_param_ptr  param);
```

`nterms`	Number of points (or terminals).
`terms`	Input point coordinates $(x_1, y_1, x_2, y_2, \ldots)$.
`length`	Length of computed SMT.
`nsps`	Number of Steiner points.
`sps`	Steiner point coordinates.
`edges`	Edges of SMT (terminals have indices 0 to `nterms`-1 while Steiner points have indices `nterms` and up).
`status`	Solution status code (see page 114).
`param`	Parameter set (`NULL`=default parameters).

Returns value zero if an SMT was computed and non-zero otherwise.

An example is given in Section 2.1.

gst_hgmst

Given an edge-weighted hypergraph, construct a minimum spanning tree (MST) in this hypergraph.

Any of the output parameters may be set to `NULL` if the corresponding output is not needed. It is the responsibility of the user to allocate sufficient memory for the output arrays.

```
int gst_hgmst (int            nverts,
               int            nedges,
               int*           edge_sizes,
               int*           edges,
               double*        weights,
               double*        length,
               int*           nmstedges,
               int*           mstedges,
               int*           status,
               gst_param_ptr  param);
```

`nverts`	Number of vertices in the hypergraph.
`nedges`	Number of edges in the hypergraph.
`edge_sizes`	Array giving number of vertices in each edge
`edges`	Array of vertices contained in each edge.
`weights`	Array of edge weights.
`nmstedges`	Number of edges in the minimum spanning tree.
`mstedges`	Array of edges contained in the minimum spanning tree.
`status`	Solution status code (see page 114).
`param`	Parameter set (`NULL`=default parameters).

Returns value zero if an MST was computed and non-zero otherwise.

Example:

```
static int edge_sizes [] = {2, 2, 2, 3};
static int edges [] = {0, 1,        /* edge 0 */
                       0, 2,        /* edge 1 */
                       1, 2,        /* edge 2 */
                       0, 1, 2};    /* edge 3 */
static double weights [] = {3.0, 2.0, 1.0, 4.0};
double length;
int code, i, nmstedges, mstedges [2];

code = gst_hgmst (3,       /* nverts */
                  4,       /* nedges */
                  edge_sizes,
                  edges,
                  weights,
                  &length,
                  &nmstedges,
                  mstedges,
                  NULL,    /* ignore status */
                  NULL);   /* use default parameters */
if (code != 0) {
        fprintf (stderr, "Return code = %d\n", code);
        exit (1);
}
printf ("Optimal solution = %g: ", length);
for (i = 0; i < nmstedges; i++) {
        printf (" %d", mstedges [i]);
}
printf ("\n");
```

3.6 Parameter setting and querying functions

A parameter set is an object that holds values for all parameters in the library. The library provides the following operations on parameter sets:

- create a parameter set having "default" values,
- change parameter settings in a parameter set,
- query the current, default, minimum and maximum values of any parameter,
- query the type of a parameter,
- copy an existing parameter set,
- free a parameter set.

Parameter sets have type `gst_param_ptr`. Various library functions require a parameter set to be provided as an argument. In all such cases it is valid for the caller to pass a `NULL` pointer, in which case default settings will be used for all parameters.

Each supported parameter has a specific type. When querying the type of a parameter, the library responds with an integer value that denotes the corresponding parameter type. The parameter types supported, together with the integer values that denote them are as follows:

Type	**Macro Name**	**Value**
`int`	`GST_PARAMTYPE_INTEGER`	1
`double`	`GST_PARAMTYPE_DOUBLE`	2
`char*`	`GST_PARAMTYPE_STRING`	3
`gst_channel_ptr`	`GST_PARAMTYPE_CHANNEL`	4

Externally each parameter has a unique number defined by a `GST_PARAM` macro (see Appendix A). This macro is used as an argument to the parameter get/set functions. Note that there are distinct parameter get/set functions for each parameter type.

gst_create_param

Create a new parameter set with default parameters.

```
gst_param_ptr gst_create_param (int*  status);
```

`status`	Status code (zero if operation was successful and non-zero otherwise).

Returns new parameter set with default parameters.

Example:

```
int status;

/* Create a default parameter set */
gst_param_ptr myparam = gst_create_param(&status);

/* Change one parameter to a non-default value */
gst_set_int_param(myparam, GST_MAX_FST_SIZE, 4);

/* Use the new parameter set...*/
```

gst_copy_param

Copy all parameter values from one parameter set into another.

```
int gst_copy_param (gst_param_ptr  dst,
                    gst_param_ptr  src);
```

`dst`	Parameter set that should be overwritten. If `NULL`, this routine does nothing.
`src`	Parameter set that should be copied. A `NULL` pointer is handled as the default set of parameters.

Returns zero if the parameter set was copied successfully.

Example:

```
/* Assume that param1 is an existing parameter set */

gst_param_ptr param2 = gst_create_param(NULL);
if (gst_copy_param (param2, param1)) {
   printf("Could not copy parameter set.\n");
   exit(1);
}

/* At this point param2 is a copy of param1 */
```

gst_free_param

Free parameter set. Freeing a parameter set that is still referenced by any other GeoSteiner object (e.g., by a problem solution state object) produces undefined behavior.

```
int gst_free_param (gst_param_ptr  param);
```

`param`	Parameter set that should be freed. If `NULL`, this routine does nothing.

Returns zero if the parameter set was freed successfully.

Example:

```
/* Free existing parameter set myparam */
gst_free_param(myparam);
```

gst_set_dbl_param

Change value of a specified double parameter in a given parameter set.

```
int gst_set_dbl_param (gst_param_ptr  param,
                       int            whichparam,
                       double         newvalue);
```

`param`	Parameter set.
`whichparam`	Parameter ID of double parameter to modify (`GST_PARAM` macro).
`newvalue`	New value for this parameter.

Returns zero if the parameter was set successfully.

Example:

```
/* Set a CPU time limit of 0.5 seconds for parameter set myparam */
gst_set_dbl_param(myparam, GST_PARAM_CPU_LIMIT, 0.5);
```

gst_get_dbl_param

Get current value of a specified double parameter from a given parameter set.

```
int gst_get_dbl_param (gst_param_ptr  param,
                       int            whichparam,
                       double*        value);
```

`param`	Parameter set.
`whichparam`	Parameter ID of double parameter to access (`GST_PARAM` macro).
`value`	Current value of parameter (pointer to double variable).

Returns zero if the parameter was accessed successfully.

Example:

```
double cpulimit;
gst_get_dbl_param(myparam, GST_PARAM_CPU_LIMIT, &cpulimit);
printf ("The current CPU time limit is %.2f.\n", cpulimit);
```

gst_query_dbl_param

Query properties of a specified double parameter in a given parameter set.

```
int gst_query_dbl_param (gst_param_ptr  param,
                         int            whichparam,
                         double*        current_value,
                         double*        default_value,
                         double*        min_value,
                         double*        max_value);
```

`param`	Parameter set.
`whichparam`	Parameter ID of double parameter to query (`GST_PARAM` macro).
`current_value`	Current value of parameter (pointer to double variable).
`default_value`	Default value of parameter (pointer to double variable).
`min_value`	Minimum value of parameter (pointer to double variable).
`max_value`	Maximum value of parameter (pointer to double variable).

Each of the last four arguments may be `NULL` if the corresponding value is not needed.

Returns zero if the parameter was queried successfully.

Example:

```
/* myparam is an existing parameter set */
double curval, defval, minval, maxval;
if (gst_query_dbl_param (myparam,
                         GST_PARAM_GAP_TARGET,
                         &curval,
                         &defval,
                         &minval,
                         &maxval) != 0) {
        fprintf (stderr, "Parameter query failed.\n");
        exit (1);
}
printf ("Gap target: current=%g, default=%g, min=%g, max=%g.\n",
        curval, defval, minval, maxval);
```

gst_set_int_param

Change value of a specified integer parameter in a given parameter set.

```
int gst_set_int_param (gst_param_ptr  param,
                       int            whichparam,
                       int            newvalue);
```

`param`	Parameter set.
`whichparam`	Parameter ID of integer parameter to modify (`GST_PARAM` macro).
`newvalue`	New value for this parameter.

Returns zero if the parameter was set successfully.

Example:

```
/* Collect the 10 best solutions. */
gst_set_int_param (myparam, GST_PARAM_NUM_FEASIBLE_SOLUTIONS, 10);
```

gst_get_int_param

Get current value of a specified integer parameter from a given parameter set.

```
int gst_get_int_param (gst_param_ptr  param,
                       int            whichparam,
                       int*           value);
```

`param`	Parameter set.
`whichparam`	Parameter ID of integer parameter to access (`GST_PARAM` macro).
`value`	Current value of parameter (pointer to integer variable).

Returns zero if the parameter was accessed successfully.

Example:

```
int vlimit;
gst_get_int_param(myparam, GST_PARAM_BACKTRACK_MAX_VERTS, &vlimit);
printf ("The current backtrack search vertex limit is %d.\n", vlimit);
```

gst_query_int_param

Query properties of a specified integer parameter in a given parameter set.

```
int gst_query_int_param (gst_param_ptr  param,
                         int            whichparam,
                         int*           current_value,
                         int*           default_value,
                         int*           min_value,
                         int*           max_value);
```

`param`	Parameter set.
`whichparam`	Parameter ID of integer parameter to query (`GST_PARAM` macro).
`current_value`	Current value of parameter (pointer to integer variable).
`default_value`	Default value of parameter (pointer to integer variable).
`min_value`	Minimum value of parameter (pointer to integer variable).
`max_value`	Maximum value of parameter (pointer to integer variable).

Each of the last four arguments may be `NULL` if the corresponding value is not needed.

Returns zero if the parameter was queried successfully.

Example:

```
/* param is an existing parameter set */
int curval, defval, minval, maxval;
if (gst_query_int_param (param,
                         GST_PARAM_BRANCH_VAR_POLICY,
                         &curval,
                         &defval,
                         &minval,
                         &maxval) != 0) {
        fprintf (stderr, "Parameter query failed.\n");
        exit (1);
}
printf ("Branch variable policy: "
        "current=%g, default=%g, min=%g, max=%g.\n",
        curval, defval, minval, maxval);
```

gst_set_str_param

Change value of a specified string parameter in a given parameter set.

```
int gst_set_str_param (gst_param_ptr  param,
                       int            whichparam,
                       const char*    str);
```

`param`	Parameter set.
`whichparam`	Parameter ID of string parameter to access (`GST_PARAM` macro).
`chan`	New value for this parameter.

Returns zero if the parameter was set successfully.

Example:

```
/* Establish a name for my problem instance. */
#define MY_INSTANCE_NAME_PARAM  -123
int code;
code = gst_set_str_param (myparam,
                          MY_INSTANCE_NAME_PARAM,
                          "Bowser");
if (code != 0) {
        fprintf (stderr, "gst_set_str_param failed.\n");
        exit (1);
}
```

gst_get_str_param

Get current value of a specified string parameter in a given parameter set.

```
int gst_get_str_param (gst_param_ptr  param,
                       int            whichparam,
                       int*           length,
                       char*          str);
```

`param`	Parameter set.
`whichparam`	Parameter ID of string parameter to access (`GST_PARAM` macro).
`length`	The length of the string is written to this integer (unless it is a `NULL` pointer). A length of `-1` indicates that the parameter has the value `NULL`, which is distinct from a string of length zero.
`str`	The current value for this parameter is copied to the string provided here (unless it is a `NULL` pointer).

Returns zero if the parameter was accessed successfully.

Example:

```
#define MY_INSTANCE_NAME_PARAM  -123
int code, length;
char* value;

/* First, get length of the string. */
gst_set_str_param (myparam,
                   MY_INSTANCE_NAME_PARAM,
                   &length,
                   NULL);
value = NULL;
if (length >= 0) {
        /* Allocate buffer to receive string value. */
        value = (char *) malloc (length + 1);
        code = gst_set_str_param (myparam,
                                  MY_INSTANCE_NAME_PARAM,
                                  NULL,
                                  value);
}
printf ("My problem instance name = %s\n",
        (value == NULL) ? "<null>" : value);
if (value != NULL) {
        free (value);
}
```

gst_set_chn_param

Change value of a specified channel parameter in a given parameter set.

```
int gst_set_chn_param (gst_param_ptr    param,
                       int              whichparam,
                       gst_channel_ptr  chan);
```

`param`	Parameter set.
`whichparam`	Parameter ID of a channel parameter to modify (`GST_PARAM` macro).
`chan`	New value for this parameter.

Returns zero if the parameter was set successfully.

Example:

```
int code;
gst_channel_ptr chan;

/* Create a channel directed to stdout. */
chan = gst_create_channel (NULL, NULL);
gst_channel_add_file (chan, stdout, NULL);

/* Direct solver trace info to stdout. */
code = gst_set_chn_param (myparam, GST_PARAM_PRINT_SOLVE_TRACE, chan);
if (code != 0) {
        fprintf (stderr, "gst_set_chn_param failed.\n");
        exit (1);
}
```

gst_get_chn_param

Get current value of a specified channel parameter from a given parameter set.

```
int gst_get_chn_param (gst_param_ptr     param,
                       int               whichparam,
                       gst_channel_ptr*  chan);
```

`param`	Parameter set.
`whichparam`	Parameter ID of channel parameter to access (`GST_PARAM` macro).
`chan`	Current value for this parameter (pointer to channel variable).

Returns zero if the parameter was accessed successfully.

Example:

```
int code;
gst_channel_ptr chan;

/* Get current solver trace channel. */
code = gst_get_chn_param (myparam,
                          GST_PARAM_PRINT_SOLVE_TRACE,
                          &chan);
if (code != 0) {
        fprintf (stderr, "gst_get_chn_param failed.\n");
        exit (1);
}
if (chan != NULL) {
        /* Turn off the trace and destroy the channel. */
        gst_set_chn_param (myparam,
                           GST_PARAM_PRINT_SOLVE_TRACE,
                           NULL);
        gst_free_channel (chan);
}
```

gst_get_param_id

Translate a parameter name into the corresponding parameter id.

```
int gst_get_param_id (const char*       param_name,
                      int*              param_id);
```

`param_name`	The name of a parameter (e.g., ”max_fst_size”, or ”GST_PARAM_MAX_FST_SIZE”).
`param_id`	Address of an integer to store the parameter ID corresponding to the given parameter name. This will be -1 for unknown or unrecognizable parameter names. The `param_id` argument can be `NULL`, if the actual parameter ID value is not required.

Returns zero if the `param_name` was recognized and the parameter ID was successfully found.

Example:

```
int parmid;
if (gst_get_param_id ("save_format", &parmid) != 0) {
        fprintf (stderr, "gst_get_param_id failed.\n");
        exit (1);
}
printf ("Parameter ID: %d\n", parmid);
```

gst_get_param_type

Get the type of a specified parameter id.

```
int gst_get_param_type (int   whichparam,
                        int*  type);
```

`whichparam`	Parameter ID to query (`GST_PARAM` macro).
`type`	This integer is set to the type of the parameter. The parameter types and their encodings as integer values are given in the table on page 35.

Returns zero if the type was found successfully.

Example:

```
char* str;
int parmtype;
if (gst_get_param_type (GST_PARAM_SAVE_FORMAT, &parmtype) != 0) {
        fprintf (stderr, "gst_get_param_type failed.\n");
        exit (1);
}
switch (parmtype) {
        case GST_PARAMTYPE_INTEGER:  str = "int";     break;
        case GST_PARAMTYPE_DOUBLE:   str = "double";  break;
        case GST_PARAMTYPE_STRING:   str = "string";  break;
        case GST_PARAMTYPE_CHANNEL:  str = "channel"; break;
        default:                     str = "unknown"; break;
}
printf ("Parameter is of type %s.\n", str);
```

gst_set_param

Set the value of a named parameter from the given string. This routine permits the value of any integer, double or string parameter to be set to the value given in text string form. This is a convenient way to set parameters from command line arguments.

```
int gst_set_param (gst_param_ptr  param,
                   const char*    name,
                   const char*    value);
```

`param`	Parameter set.
`name`	Name of parameter to set (see Appendix A).
`value`	Text string containing data value to set.

Example:

```
int main (int argc, char **argv)
{
int             i, j;
char *          ap;
gst_channel_ptr myparm;

        gst_open_geosteiner (NULL);
        myparam = gst_create_param (NULL);

        /* Parse arguments such as: -ZBRANCH_VAR_POLICY 3 */
        for (i = 1; i < argc; i++) {
                ap = argv [i];
                if ((ap[0] != '-') || (ap[1] != 'Z')) usage ();
                j = gst_set_param (myparam, &ap[2], argv [i+1]);
                if (j != 0) usage ();
                ++i;
        }
        /* Parameters are now set... */
}
```

3.7 Metric setting and querying functions

The support of different metrics in the GeoSteiner library is primarily handled by metric objects. Some functions in the library use these metric objects automatically, e.g., **gst_esmt()**, while others require one to specify a metric object, e.g., **gst_smt()**. The metric objects provide a simple way to make general applications support several different metrics. An example of this can be found in the demo program `demo2.c` which is the code for a small program supporting all metrics supported by GeoSteiner.

Two L_p-metrics, L_1 (rectilinear) and L_2 (Euclidean), are supported. Also, all uniform metrics — so-called λ-metrics — are supported. The latter are metrics where only a limited number $\lambda \geq 2$ of equally-spaced orientations are allowed for the edges in a solution. For $\lambda = 2$ this is identical to the rectilinear metric, L_1.

When a metric object has been created, the distance between two points in the metric can be obtained by calling **gst_distance()**. This is especially useful for the λ-metrics for which efficient calculation is non-trivial.

The following macros are used for identifying the supported metrics:

Metric Type	**Macro Name**	**Value**
None	`GST_METRIC_NONE`	`0`
L_p	`GST_METRIC_L`	`1`
Uniform	`GST_METRIC_UNIFORM`	`2`

gst_create_metric

A metric is defined by a type and a parameter. For the L_p-metric this parameter p must be either 1 or 2, and for the λ-metric we must have $\lambda \geq 2$.

Note that even though the L_1-metric and the λ-metric with parameter 2 are the same (rectilinear metric), you cannot expect them to give exactly the same results when used to solve Steiner problems. The first one will result in the use of a dedicated FST generator for the rectilinear problem and the latter will result in the use of a general FST generator for λ-metrics. If you are aiming for speed then use the L_1-metric.

```
gst_metric_ptr gst_create_metric (int   type,
                                  int   parameter,
                                  int*  status);
```

`type`	Metric type (see macro values in the table on page 55).
`parameter`	Metric parameter.
`status`	Status code (zero if operation was successful and non-zero otherwise).

Returns new metric object.

Example:

```
/* Creating a Euclidean metric object */
gst_metric_ptr metric;
metric = gst_create_metric (GST_METRIC_L, 2, NULL);

/* And use it as a parameter to gst_smt */
gst_smt (nterms, terms, &length, NULL, NULL, NULL, NULL, NULL,
         metric, NULL);
```

gst_free_metric

Free an existing metric object. Freeing a metric object that is still referenced by any other GeoSteiner object (e.g., a hypergraph object) produces undefined behavior.

```
int gst_free_metric (gst_metric_ptr  metric);
```

`metric`	Metric object. Does nothing if `NULL`.

Returns zero if operation was successful.

Example:

```
/* Free parameter object mymetric */
gst_free_metric (mymetric);
```

gst_copy_metric

Copy attributes from one metric object to another.

```
int gst_copy_metric (gst_metric_ptr  dst,
                     gst_metric_ptr  src);
```

`dst`	Metric object that should be overwritten.
`src`	Metric that should be copied. A `NULL` pointer is considered as a ”None” metric type (see table on page 55).

Returns zero if metric object was copied.

Example:

```
gst_metric_ptr newmetric;

newmetric = gst_create_metric (GST_METRIC_NONE, 0);

gst_copy_metric (newmetric, oldmetric);

/* newmetric is now the same metric as oldmetric. */
```

gst_distance

Compute the distance between two points under a given metric.

```
double gst_distance (gst_metric_ptr  metric,
                     double          x1,
                     double          y1,
                     double          x2,
                     double          y2);
```

`metric`	Metric object.
`x1`	X-coordinate for first point.
`y1`	Y-coordinate for first point.
`x2`	X-coordinate for second point.
`y2`	Y-coordinate for second point.

Returns the distance. Returned value is always zero if metric type is ”None”.

Example:

```
/* Assume that mymetric is a metric object. */
/* Compute distance between points (0,0) and (1,1). */
double d;
d = gst_distance (mymetric, 0.0, 0.0, 1.0, 1.0);
```

gst_get_metric_info

Get the information about a metric object.

```
int gst_get_metric_info (gst_metric_ptr  metric,
                         int*            type,
                         int*            parameter);
```

`metric`	Metric object.
`type`	A pointer to an integer in which to place the metric type. See the possible types in the table on page 55.
`parameter`	An optional pointer to an integer in which to place the metric parameter. See the possible parameters in the description of **gst_create_metric()**.

Returns zero if operation was successful. Either of the last two arguments may be `NULL` if the corresponding value is not needed.

Example:

```
/* Let mymetric be a metric object */
int type, parameter;
gst_get_metric_info (mymetric, &type, &parameter);
switch (type) {
        case GST_METRIC_NONE:
                printf ("Metric is None.\n");
                break;
        case GST_METRIC_L:
                printf ("Metric is L%d.\n", parameter);
                break;
        case GST_METRIC_UNIFORM:
                printf ("Metric is Uniform %d.\n", parameter);
                break;
        default:
                printf ("Metric is unknown!\n");
}
```

3.8 Property list setting and querying functions

Property lists can be used to hold values which are rarely updated (the data structure holding the information *cannot* be queried/updated in constant time). The following basic operations are provided by the library:

- create an empty property list,
- set/create a value in a property list,
- delete a value from a property list,
- get a value in a property list,
- query the type of a property,
- copy a property list,
- free a property list (including its content).

A property list has type `gst_proplist_ptr` and a property is known by its property ID (a macro name which expands to a signed integer).

The main purpose of property lists is to make extra information about the solution process available to the user through a simple interface. Any property ID with a value larger than or equal to zero is reserved by the library. Negative values can be freely used by the user. The property ID values (and their macro names) currently in use can be found in Appendices B and C.

Note that there are distinct property get/set functions for different property types. The type of a given property — which is an integer — can be queried. The supported property types, together with the integer values that denote them are as follows:

Type	**Macro Name**	**Value**
`int`	`GST_PROPTYPE_INTEGER`	1
`double`	`GST_PROPTYPE_DOUBLE`	2
`char*`	`GST_PROPTYPE_STRING`	3

gst_create_proplist

Create a new empty property list.

```
gst_proplist_ptr
    gst_create_proplist (int*  status);
```

`status`	Status code (zero if operation was successful and non-zero otherwise). May be `NULL` if value is not needed.

Returns new property list.

Example:

```
gst_proplist_ptr plist;
int status;
plist = gst_create_proplist (&status);
if (status != 0) {
        fprintf (stderr, "Unable to create property list.\n");
        exit (1);
}
gst_set_int_property (plist, GST_PROP_SOLVER_ROOT_OPTIMAL, 1);
```

gst_free_proplist

Free an existing property list. Freeing a property list that is still referenced by existing GeoSteiner objects (e.g., hypergraphs and solvers) results in undefined behavior. In most cases it is an error to free a property list that was not obtained via a call to **gst_create_proplist()**.

```
int gst_free_proplist (gst_proplist_ptr  plist);
```

`plist`	A property list to free. If `NULL`, this routine does nothing.

Returns a status code (zero if operation was successful and non-zero otherwise).

Example:

```
gst_proplist_ptr plist;
plist = gst_create_proplist (NULL);

/* Various operations on plist... */

gst_free_proplist (plist);
```

gst_copy_proplist

Empty the destination property list and copy all properties into it from the source property list.

```
int gst_copy_proplist (gst_proplist_ptr  dst,
                       gst_proplist_ptr  src);
```

`dst`	Property list that should be overwritten.
`src`	Property list that should be copied. A `NULL` pointer is handled as an empty property list.

Returns zero if the property list was copied successfully.

Example:

```
/* We assume that H is a hypergraph... */
gst_proplist_ptr copy;

copy = gst_create_proplist (NULL);

if (gst_copy_proplist (copy, gst_get_hg_properties(H)) == 0) {
   /* We have now created a copy of the property list for H */
}
else {
   /* Something went wrong */
}

/* Use new copy of property list... */

/* Free copy */
gst_free_proplist (copy);
```

gst_get_property_type

Query the type of a given property.

```
int gst_get_property_type (gst_proplist_ptr  plist,
                           int               propid,
                           int*              type);
```

`plist`	An existing property list.
`propid`	A property ID value.
`type`	Pointer to an integer which will be overwritten with the type of the property.

Return a status code (zero if operation was successful and non-zero otherwise).

Example:

```
/* We assume that H is a hypergraph... */
int type;

if (gst_get_property_type (gst_get_hg_properties(H),
                           GST_PROP_HG_GENERATION_TIME,
                           &type) != 0) {
   /* Something went wrong */
}
else {
   switch (type) {
   case GST_PROPTYPE_INTEGER: /* Property is an integer value */
      break;
   case GST_PROPTYPE_DOUBLE:  /* Property is a floating point value */
      break;
   case GST_PROPTYPE_STRING:  /* Property is a string value */
      break;
   default: /* Something went wrong */
   }
}
```

gst_delete_property

Remove any value that might be defined for the given property ID, regardless of type.

```
int gst_delete_property (gst_proplist_ptr   plist,
                         int                propid);
```

`plist`	Property list.
`propid`	ID of property to delete.

Returns zero if the property was successfully deleted from the property list.
Returns `GST_ERR_INVALID_PROPERTY_LIST` if the property list itself is invalid.
Returns `GST_ERR_PROPERTY_NOT_FOUND` if no property having the given ID exists.

Example:

```
/* We are given a property list plist */
#define MY_PROPERTY_ID  -1000

gst_delete_property (plist, MY_PROPERTY_ID);

/* plist no longer has any value defined */
/* for property ID -1000. */
```

gst_get_dbl_property

Get the value of a specified double property from a given property list. The specified property must be of type double or an error is returned. ID values greater than or equal to zero are reserved for GeoSteiner's use. Negative ID values can be freely used by user applications.

```
int gst_get_dbl_property (gst_proplist_ptr  plist,
                          int               propid,
                          double*           value);
```

`plist`	Property list.
`propid`	ID of double property to retrieve.
`value`	Current value of property (pointer to double variable). May be `NULL` if value is not needed.

Returns zero if the property was accessed successfully.
Returns `GST_ERR_PROPERTY_NOT_FOUND` if no property having the given ID exists.
Returns `GST_ERR_PROPERTY_TYPE_MISMATCH` if the property exists but does not have type double.

Example:

```
/* We are given a property list plist and a double value has
   been set for the ID value GST_PROP_USER_MYVALUE */
#define GST_PROP_USER_MY_DBL_VALUE   -1000

double value;
gst_get_dbl_property (plist,
                      GST_PROP_USER_MY_DBL_VALUE,
                      &value);
printf ("My_dbl_value is currently set at %.2f.\n", value);
```

gst_get_int_property

Get the value of a specified property from the given property list. The specified property must be of type integer or an error is returned. ID values greater than or equal to zero are reserved for GeoSteiner's use. Negative ID values can be freely used by user applications.

```
int gst_get_int_property (gst_proplist_ptr  plist,
                          int               propid,
                          int*              value);
```

`plist`	Property list.
`propid`	ID of integer property to retrieve.
`value`	Current value of property (pointer to integer variable). May be `NULL` if value is not needed.

Returns zero if the property was accessed successfully.
Returns `GST_ERR_PROPERTY_NOT_FOUND` if no property having the given ID exists.
Returns `GST_ERR_PROPERTY_TYPE_MISMATCH` if the property exists but does not have type integer.

Example:

```
/* We are given a property list plist and an integer value has
   been set for the ID value GST_PROP_USER_MY_INT_VALUE */
#define GST_PROP_USER_MY_INT_VALUE   -1001

int value;
gst_get_int_property (plist,
                      GST_PROP_USER_MY_INT_VALUE,
                      &value);
printf ("My_int_value is currently set at %d.\n", value);
```

gst_get_str_property

Get the value of a specified property from the given property list. The specified property must be of type string or an error is returned. ID values greater than or equal to zero are reserved for GeoSteiner's use. Negative ID values can be freely used by user applications.

```
int gst_get_str_property (gst_proplist_ptr  plist,
                          int               propid,
                          int*              length,
                          char*             str);
```

`plist`	Property list.
`propid`	ID of string property to retrieve.
`length`	The length of the string is written to this integer (unless it is a `NULL` pointer). The returned length does *not* include the terminating null character. The returned length is -1 if the property value is a `NULL` pointer (which is distinct from a zero length string).
`str`	The current value for this parameter is copied into the buffer provided here (unless it is a `NULL` pointer).

Returns zero if the property was accessed successfully.
Returns `GST_ERR_PROPERTY_NOT_FOUND` if no property having the given ID exists.
Returns `GST_ERR_PROPERTY_TYPE_MISMATCH` if the property exists but does not have type string.

Example:

```
int code, length;
char* buf;
buf = NULL;
code = gst_get_str_property (plist, GST_PROP_HG_NAME,
                             &length, NULL);
if ((code == 0) && (length >= 0)) {
        buf = (char *) malloc (length + 1);
        gst_get_str_property (plist,
                              GST_PROP_HG_NAME,
                              NULL,
                              buf);
}
printf ("Hypergraph name is %s\n",
        (buf == NULL) ? "<NULL>" : buf);
if (buf != NULL) free (buf);
```

gst_get_properties

Retrieve all property IDs and their types from the given property list.

```
int gst_get_properties (gst_proplist_ptr  plist,
                        int*              count,
                        int*              propids,
                        int*              types);
```

`plist`	Property list.
`count`	Number of properties in the given `plist` (unless it is a `NULL` pointer).
`propids`	Buffer to receive the property IDs of each property in `plist` (unless it is a `NULL` pointer).
`types`	Buffer to receive the types of each property in `plist` (unless it is a `NULL` pointer).

Returns zero if the properties were successfully retrieved.

Example:

```
int count;
int* propids;
int* types;
code = gst_get_properties (plist, &count, NULL, NULL);
if (code != 0) {
       /* Something went wrong. */
       exit (1);
}
propids = (int *) malloc (count * sizeof (int));
types   = (int *) malloc (count * sizeof (int));
gst_get_properties (plist, NULL, propids, types);
for (i = 0; i < count; i++) {
        printf ("Propid = %d, type = %d.\n",
                propids [i], types [i]);
}
free (types);
free (propids);
```

gst_set_dbl_property

Change or create a specified property in the given property list. The property is added to the list if not already present. If the property already exists, its type is forced to be double. It is *legal* to do this with any property list.

```
int gst_set_dbl_property (gst_proplist_ptr  plist,
                          int               propid,
                          double            value);
```

plist	Property list.
propid	ID of double property to create or modify.
newvalue	New value for this property.

Returns zero if the property was set successfully.

Example:

```
/* Assume we are given a property list plist */
#define GST_PROP_USER_MY_DBL_VALUE   -1000

gst_set_dbl_property (plist, GST_PROP_USER_MY_DBL_VALUE, 2.71828);
```

gst_set_int_property

Change or create a a specified property in the given property list. The property is added to the list if not already present. If the property already exists, its type is forced to be integer. It is *legal* to do this with any property list.

```
int gst_set_int_property (gst_proplist_ptr  plist,
                          int               propid,
                          int               value);
```

`plist`	Property list.
`propid`	ID of integer property to create or modify.
`newvalue`	New value for this property.

Returns zero if the property was set successfully.

Example:

```
/* Assume we are given a property list plist */
#define GST_PROP_USER_MY_INT_VALUE   -1001

gst_set_int_property (plist, GST_PROP_USER_MY_INT_VALUE, 42);
```

gst_set_str_property

Change or create a specified property in the given property list. The property is added to the list if not already present. If the property already exists, its type is forced to be string. It is *legal* to do this with any property list.

```
int gst_set_str_property (gst_proplist_ptr  plist,
                          int               propid,
                          const char*       value);
```

plist	Property list.
propid	ID of string property to create or modify.
newvalue	New value for this property.

Returns zero if the property was set successfully.

Example:

```
/* Assume we are given a property list plist */
gst_set_str_property (plist, GST_PROP_HG_NAME, "Oobleck");
```

3.9 Hypergraph functions

The hypergraph object represents an arbitrary hypergraph that can be decorated with a variety of additional (and optional) data. For example, the edges can be given weights. In general, the goal of GeoSteiner is to find a spanning tree of minimum total weight using the edges of the hypergraph.

In this section we document all of the operations provided for creating, destroying and manipulating hypergraph objects.

Hypergraphs can be embedded in the plane: Vertices can be given coordinates and hyperedges can be associated with trees in the plane. Also, every hypergraph has an associated metric object (Section 3.7), a scaling object (Section 3.14) and a property list (Section 3.8).

The library interfaces have been designed to permit maximum flexibility in using the various operations provided. For example, it is intended that the user be able to define a hypergraph, solve it, modify some attributes of the hypergraph (e.g., change some of the edge costs), and re-solve the modified problem. The library should be smart enough to know when the problem can be re-solved starting from the most recent solution state — and when it is necessary to discard the previous solution state and re-solve the current problem from scratch.

gst_create_hg

Create an instance of an empty hypergraph. The hypergraph initially has no vertices and no edges. After creating an empty hypergraph, the next step is normally to give it the desired number of vertices using **gst_set_hg_number_of_vertices()**, and then add the edges using **gst_set_hg_edges()**. Doing the steps in this order avoids the failure that would result from attempting to add edges that refer to non-existent vertices.

```
gst_hg_ptr gst_create_hg (int*  status);
```

`status`	Status code (zero if the operation was successful and non-zero otherwise). May be `NULL` if the value is not needed.

Returns new hypergraph object.

Example:

```
gst_hg_ptr h;
int status;
h = gst_create_hg (&status);
if (status != 0) {
        /* Something went wrong */
}
/* Make it be a complete hypergraph on 3 vertices */
status = gst_set_hg_number_of_vertices (h, 3);
if (status != 0) {
        /* Error */
}
else {
        static int edge_sizes [] = {2, 2, 2, 3};
        static int edges [] = {0, 1, 0, 2, 1, 2, 0, 1, 2};
        status = gst_set_hg_edges (h, 4, edge_sizes, edges, NULL);
}
```

gst_copy_hg

Make a copy of a given hypergraph. Any data associated with the destination hypergraph is discarded, and the following attributes are copied from the source hypergraph (if present): vertices, edges, edge weights, metric object info, scale object info, property list, vertex embedding, and edge embedding.

```
int gst_copy_hg (gst_hg_ptr  dst,
                 gst_hg_ptr  src);
```

`dst`	Destination hypergraph object. All existing data in the destination is discarded.
`src`	Source hypergraph object to copy.

Returns zero if the hypergraph was copied successfully.

Example:

```
/* Assume that h is an existing hypergraph */
gst_hg_ptr newhg;
newhg = gst_create_hg (NULL);
status = gst_copy_hg (newhg, h);
if (status != 0) {
        fprintf (stderr, "Error copying hypergraph\n");
        exit (1);
}
gst_set_hg_edge_weights (newhg, NULL);
/* newhg is now a copy of h, but with all edge weights = 1. */
```

gst_copy_hg_edges

Make a copy of a given hypergraph with a subset of the original edges. Any data associated with the destination hypergraph is discarded, and the following attributes are copied from the source hypergraph (if present): vertices, (subset of) edges, (subset of) edge weights, metric object info, scale object info, property list, vertex embedding, and edge embedding.

```
int gst_copy_hg_edges (gst_hg_ptr  dst,
                       gst_hg_ptr  src,
                       int         nedges,
                       int*        edges);
```

`dst`	Destination hypergraph object. All existing data in the destination is discarded.
`src`	Source hypergraph object to copy.
`nedges`	Number of edges to copy from source hypergraph.
`edges`	Index values of edges to copy from source hypergraph.

Returns zero if (a subset of) the hypergraph was copied successfully.

Example:

```
/* Assume that h is an existing hypergraph with 10 edges */
static int edges [] = {2, 4, 6, 8};
gst_hg_ptr newhg;
newhg = gst_create_hg (NULL);
status = gst_copy_hg_edges (newhg, h, 4, edges);
if (status != 0) {
        fprintf (stderr, "Error copying hypergraph\n");
        exit (1);
}
/* newhg is now a copy of h but having only 4 of the edges of h */
```

gst_free_hg

Remove a hypergraph and free all associated memory, including associated properties.

```
int gst_free_hg (gst_hg_ptr  H);
```

`H`	Hypergraph to free. If `NULL`, this function does nothing.

Returns zero if the hypergraph was freed successfully.

Example:

```
/* Assume that h is an existing hypergraph */
int status;
status = gst_free_hg (h);
if (status != 0) {
        fprintf (stderr, "Error freeing hypergraph\n");
        exit (1);
}
```

gst_set_hg_number_of_vertices

Define the number of vertices of a hypergraph.

```
int gst_set_hg_number_of_vertices (gst_hg_ptr  H,
                                   int         nverts);
```

`H`	Hypergraph.
`nverts`	Number of vertices `H` should have (non-negative number).

Returns zero if the number of vertices was set successfully.

Example:

```
/* Construct a hypergraph with 20 vertices (no error checking) */
gst_hg_ptr hg;

hg = gst_create_hg (NULL);
gst_set_hg_number_of_vertices (hg, 20);
```

gst_set_hg_edges

Define the set of edges of a hypergraph (default associated information).

```
int gst_set_hg_edges (gst_hg_ptr  H,
                      int         nedges,
                      int*        edge_sizes,
                      int*        edges,
                      double*     weights);
```

`H`	Hypergraph.
`nedges`	Number of edges `H` should have.
`edge_sizes`	Number of vertices for each edge.
`edges`	Vertex indices of each edge.
`weights`	Edge weights (if `NULL` then all edge weights are 1).

Returns zero if the edges were defined successfully.

Example:

```
/* Construct a complete hypergraph on 3 vertices
   with edge weights 1 (no error checking) */

gst_hg_ptr h;
static int edge_sizes [] = {2, 2, 2, 3};
static int edges [] = {0, 1, 0, 2, 1, 2, 0, 1, 2};

h = gst_create_hg (NULL);
gst_set_hg_number_of_vertices (h, 3);
gst_set_hg_edges (h, 4, edge_sizes, edges, NULL);
```

gst_set_hg_edge_weights

Set all edge weights of a hypergraph.

```
int gst_set_hg_edge_weights (gst_hg_ptr  H,
                             double*     weights);
```

`H`	Hypergraph.
`weights`	Array of edge weights of length equal to the number of edges in `H` (if `NULL` then all edge weights are set to 1).

Returns zero if the edges weights were set successfully.

Example:

```
/* Assume that h is a hypergraph with 4 edges */

static double weights [] = {1.0, 2.0, 3.0, 4.0};
int status;

status = gst_set_hg_edge_weights (h, weights);
if (status != 0) {
        fprintf (stderr, "Error setting edge weights\n");
        exit (1);
}
/* The edges of h now have weights 1, 2, 3 and 4 */
```

gst_set_hg_vertex_embedding

Embed the vertices in a hypergraph in some k-dimensional space. (In the current version only the 2-dimensional space, the plane, is supported.)

```
int gst_set_hg_vertex_embedding (gst_hg_ptr  H,
                                 int         dim,
                                 double*     coords);
```

`H`	Hypergraph whose vertices should be embedded.
`dim`	Dimension of space (currently only dimension 2 is supported).
`coords`	Vertex coordinates $(x_1, y_1, x_2, y_2, \ldots)$. Length must be the dimension times the number of vertices in the hypergraph.

Returns zero if the vertices were embedded successfully.

Example:

```
/* Assume that h is an existing hypergraph with four vertices */
static double coords [] = {0, 0, 1, 0, 1, 1, 0, 1};
int status;
status = gst_set_hg_vertex_embedding (h, 2, coords);
if (status != 0) {
        fprintf (stderr, "Error embedding vertices\n");
        exit (1);
}
/* The four vertices of h are now embedded as
   (0,0), (0,1), (1,1) and (0,1). */
```

gst_set_hg_metric

Set the metric object associated with a hypergraph.

```
int gst_set_hg_metric (gst_hg_ptr      H,
                       gst_metric_ptr  metric);
```

`H`	Hypergraph.
`metric`	Metric object that should be associated with `H` (see Section 3.7 for information on metric objects). If `NULL`, then the hypergraph metric will be set to ”None”.

Returns zero if metric was set successfully.

Example:

```
/* Assume that h is an existing hypergraph */

/* Create a Euclidean metric object */
gst_metric_ptr metric;
metric = gst_create_metric (GST_METRIC_L, 2, NULL);

/* Associate it with h */
gst_set_hg_metric (h, metric);
```

gst_set_hg_scale_info

Set the scaling information associated with a hypergraph.

```
int gst_set_hg_scale_info (gst_hg_ptr          H,
                           gst_scale_info_ptr  scinfo);
```

`H`	Hypergraph.
`scinfo`	Scaling information that should be associated with this hypergraph (see Section 3.14). If `NULL`, then no scaling is used for this hypergraph.

Returns zero if the scaling information was set successfully.

Example:

```
/* Read a set points from stdin, generate FST hypergraph
   and set scaling information */

gst_hg_ptr hg;
gst_scale_info_ptr scinfo;
int n;
double* terms;

n = gst_get_points (stdin, 0, &terms, scinfo);
hg = gst_generate_efsts (n, terms, NULL, NULL);
gst_set_hg_scale_info (hg, scinfo);
```

gst_get_hg_terminals

Get terminal vertices for a hypergraph. The terminal indices are returned in the `terms` array.

```
int gst_get_hg_terminals (gst_hg_ptr  H,
                          int*        nterms,
                          int*        terms);
```

gst_get_hg_number_of_vertices

Get the number of vertices of a hypergraph.

```
int gst_get_hg_number_of_vertices (gst_hg_ptr  H);
```

`H`	Hypergraph.

A return value of -1 implies that the hypergraph was invalid.

Example:

```
/* Assume that hg is an existing hypergraph */
int nverts;

nverts = gst_get_hg_number_of_vertices (hg);

/* nverts is now equal to the number of vertices in hg */
```

gst_get_hg_edges

Get the set of edges of a hypergraph. If any of the three final arguments is `NULL`, the corresponding information is not returned. The user has to allocate space for holding the returned data. Necessary sizes for arrays can be obtained by first obtaining the number of edges, then the edge sizes and finally the vertices for each edge (see example below).

```
int gst_get_hg_edges (gst_hg_ptr  H,
                      int*        nedges,
                      int*        edge_sizes,
                      int*        edges,
                      double*     weight);
```

`H`	Hypergraph.
`nedges`	Number of edges in this hypergraph.
`edge_sizes`	Number of vertices for each edge (pointer to an array allocated by the user).
`edges`	Vertex indices of each edges (pointer to an array allocated by the user).
`weights`	Edge weights (pointer to an array allocated by the user).

Returns zero if the edges were queried successfully.

Example:

```
/* Assume that H is some hypergraph */
int i, nedges, nedgeverts;
int* edge_sizes;
int* edges;
double* weight;

/* First we query the number of edges */
gst_get_hg_edges (H, &nedges, NULL, NULL, NULL);

/* Allocate space for edge sizes and edge weights */
edge_sizes = (int *)    malloc (nedges * sizeof (int));
weight     = (double *) malloc (nedges * sizeof (double));

/* Query edge sizes and weights */
gst_get_hg_edges (H, NULL, edge_sizes, NULL, weight);

/* Count the number of vertices in all edges */
nedgeverts = 0;
for (i = 0; i < nedges; i++)
   nedgeverts += edge_sizes[i];
edges = (int *) malloc (nedgeverts * sizeof (int));

/* Finally query vertices of edges */
gst_get_hg_edges (H, NULL, NULL, edges, NULL);
```

gst_get_hg_one_edge

Get information about one edge in the hypergraph. If any of the three last arguments to the function is `NULL`, the corresponding information is not returned.

```
int gst_get_hg_one_edge (gst_hg_ptr  H,
                         int         edge_number,
                         double*     weight,
                         int*        nverts,
                         int*        verts);
```

`H`	Hypergraph.
`edge_number`	Edge number to query (first edge is number 0).
`weight`	Weight of edge (pointer to a double variable).
`nverts`	Number of vertices in this edge (pointer to an int variable).
`terms`	Vertex indices of this edges (pointer to an array allocated by user).

Returns zero if the edge was queried successfully.

Example:

```
/* Assume that H is some hypergraph with at least 10 edges  */
int nverts;
int* verts;
double weight;

/* Query edge number 10 */
gst_get_hg_one_edge (H, 10, &weight, &nverts, NULL);

/* Allocate space for vertex indices */
verts = (int *) malloc (nverts * sizeof (int));

/* Query vertex indices */
gst_get_hg_one_edge (H, 10, NULL, NULL, verts);
```

gst_get_hg_vertex_embedding

Get the embedding of the vertices in a hypergraph.

```
int gst_get_hg_vertex_embedding (gst_hg_ptr  H,
                                 int*        dim,
                                 double*     coords);
```

`H`	Hypergraph whose vertices are embedded.
`dim`	Dimension of the space (pointer to an integer variable).
`coords`	Array in which to place the vertex coordinates of the embedding $(x_1, y_1, x_2, y_2, \ldots)$. This array must be allocated by the user, and its length must be dimension times the number of vertices in the hypergraph.

Returns zero if the embedding was returned successfully.

Example:

```
/* Assume that h is an existing hypergraph with four vertices
   embedded in the plane */
double coords[8];
int status;

status = gst_get_hg_vertex_embedding(H, NULL, coords);
if (status != 0) {
        fprintf (stderr, "Error querying vertex embedding\n");
        exit (1);
}

/* coords now holds the coordinates of the embedded vertices */
```

gst_get_hg_one_vertex_embedding

Return the embedding of a single vertex in a hypergraph.

```
int gst_get_hg_one_vertex_embedding
                          (gst_hg_ptr  H,
                           int         vertex_number,
                           double*     coords);
```

`H`	Hypergraph whose vertices are embedded.
`vertex_number`	Vertex number whose embedding should be queried (first vertex is number 0).
`coords`	Coordinates of the vertex embedding (x_1, y_1). This array must be allocated by the user, and its length equal to the dimension of the space of the embedding.

Returns zero if the embedding was returned successfully.

Example:

```
/* Assume that h is an existing hypergraph with four vertices
   embedded in the plane */
double coords[2];
int status;

/* Query embedding of vertex number 3 */
status = gst_get_hg_one_vertex_embedding(H, 3, coords);
if (status != 0) {
        fprintf (stderr, "Error querying vertex embedding\n");
        exit (1);
}

/* coords now holds the coordinates of vertex number 3 */
```

gst_get_hg_edge_embedding

Return the embedding of a subset of edges in a hypergraph. If any of the four last arguments to the function is `NULL`, the corresponding information is not returned.

```
int gst_get_hg_edge_embedding (gst_hg_ptr  H,
                               int         nhgedges,
                               int*        hgedges,
                               int*        nsps,
                               double*     sps,
                               int*        nedges,
                               int*        edges);
```

`H`	Hypergraph
`nhgedges`	Number of hyperedges that should be queried for embedding information (when equal to 0 all edges are returned).
`hgedges`	List of indices of hyperedges that should be queried. If this argument is `NULL` then the first `nhgedges` are returned.
`nsps`	Number of Steiner points in embedding of all queried hyperedges (pointer to int variable).
`sps`	Coordinates of Steiner points in the embedded hyperedges (pointer to double array allocated by user).
`nedges`	Number of edges in the *embedding* (pointer to int variable).
`edges`	Indices of the edge endpoints in *embedding* (pointer to int array allocated by user). Let n be the number of vertices in hypergraph `H`. Then hypergraph vertex endpoints have indices 0 to $n-1$ while Steiner endpoints have indices n and up.

Returns zero if the embedding was queried successfully.

Example:

```
/* Assume that H is an embedded hypergraph with
   5 vertices and 10 edges. The complete embedding
   has 15 Steiner points and 30 edges. We would like
   to get the embedding of hyperedges with even indices. */

int nsps;
int nedges;
double sps[30];
int edges[60];

static int hgedges [] = {0, 2, 4, 6, 8};

gst_get_hg_edge_embedding (H, 5, hgedges,
                           &nsps, sps, &nedges, edges);

/* Now sps contains the Steiner point coordinates,
   while edges contain edge endpoints; hypergraph
   vertices have endpoint indices 0..4 and Steiner
   points endpoint indices 5..19. */
```

gst_get_hg_one_edge_embedding

Return the embedding of a single edge in a hypergraph. Note that the indices of vertices spanned by an edge can be obtained by using **gst_get_hg_one_edge()**.

```
int gst_get_hg_one_edge_embedding
                        (gst_hg_ptr  H,
                         int         edge_number,
                         int*        nsps,
                         double*     coords,
                         int*        nedges,
                         int*        edges);
```

`H`	Hypergraph.
`edge_number`	Hyperedge number whose embedding should be queried (first hyperedge has number 0).
`nsps`	Number of Steiner points in the embedding for the hyperedge (pointer to int variable).
`coords`	Coordinates of Steiner points in embedded hyperedge (pointer to double array allocated by user).
`nedges`	Number of edges in the *embedding* (pointer to int variable).
`edges`	Indices of edge endpoints in the *embedding* (pointer to int array allocated by user). Let k be the number of vertices in the hyperedge. Then hypergraph vertex endpoints have indices 0 to $k-1$ while Steiner endpoints have indices k and up.

Returns zero if embedding was queried successfully.

Example:

```
/* Assume that H is an embedded hypergraph with 10 edges.
   We would like to get the embedding of hyperedge 7. */

int nsps;
int nedges;
double* sps;
int* edges;

gst_get_hg_one_edge_embedding (H, 7, &nsps, NULL, &nedges, NULL);

/* Allocate space */
sps   = (double *) malloc (2*nsps   * sizeof (double));
edges = (int *)    malloc (2*nedges * sizeof (int));

gst_get_hg_one_edge_embedding (H, 7, NULL, sps, NULL, edges);

/* Now sps contains the Steiner point coordinates,
   while edges contain edge endpoints. */
```

gst_get_hg_edge_status

Return the pruning status of an edge. When `gst_prune_edges` runs, it may determine that some edges are "required" (such edges *must* appear in any optimal solution). It may also determine that certain other edges are "unneeded" (at least one optimal solution exists that does not use any "unneeded" edge). By default, edges are neither "unneeded" nor "required." It is impossible for an edge to be simultaneously "unneeded" and "required."

```
int gst_get_hg_edge_status (gst_hg_ptr  H,
                            int         edge_number,
                            int*        unneeded,
                            int*        required);
```

`H`	Hypergraph.
`edge_number`	Hyperedge whose pruning status should be queried.
`unneeded`	Non-zero if edge is "unneeded" (pointer to an int variable).
`required`	Non-zero if edge is "required" (pointer to an int variable).

Returns zero if pruning status was queried successfully.

Example:

```
/* Assume that H is an embedded hypergraph with N
   edges that has been pruned.  We would like to
   get the pruning status of its edges. */

int i, unneeded, required;
const char * s;

for (i = 0; i < N; i++) {
  gst_get_hg_edge_status (H, i, &unneeded, &required);
  if (required) {
    s = "required";
  } else if (unneeded) {
    s = "unneeded";
  } else {
    s = "undecided";
  }
  printf (" Edge %d is %s\n", s);
}
```

gst_get_hg_metric

Get the metric object associated with a hypergraph.

```
int gst_get_hg_metric (gst_hg_ptr       H,
                       gst_metric_ptr*  metric);
```

`H`	Hypergraph.
`metric`	Metric object associated with this hypergraph (see Section 3.7 for information on metric objects).

Returns zero if the metric was queried successfully.

Example:

```
/* Assume that h is an existing hypergraph */

gst_metric_ptr metric;

/* Get metric associated with h */
gst_get_hg_metric (h, metric);
```

gst_get_hg_scale_info

Get the scaling information associated with a hypergraph.

```
int gst_get_hg_scale_info
                     (gst_hg_ptr           H,
                      gst_scale_info_ptr*  scinfo);
```

`H`	Hypergraph.
`scinfo`	Scaling information associated with this hypergraph (see Section 3.14).

Returns zero if the scaling information was queried successfully.

Example:

```
/* Assume that h is an existing hypergraph */

gst_scale_info_ptr scinfo;

/* Get scaling information associated with h */
gst_get_hg_scale_info (h, scinfo);
```

gst_get_hg_properties

Return the list of properties associated with a hypergraph.

```
gst_proplist_ptr
        gst_get_hg_properties (gst_hg_ptr  H);
```

H	Hypergraph

Returns the property list.

Example:

```
/* Assume we are given a hypergraph H */
double gtime, ptime;
gst_proplist_ptr hgprop;

/* Get timing information from the hypergraph, if available */
hgprop = gst_get_hg_properties (H);
gtime = 0.0; ptime = 0.0;
gst_get_dbl_property (hgprop, GST_PROP_HG_GENERATION_TIME, &gtime);
gst_get_dbl_property (hgprop, GST_PROP_HG_PRUNING_TIME, &ptime);

printf ("Generation time: %.2f\n", gtime);
printf ("Pruning time: %.2f\n", ptime);
printf ("Total time: %.2f\n", ptime + gtime);

/* We can set our own property in the same list e.g. for later use */
#define GST_PROP_USER_TOTAL_TIME       -1000
gst_set_dbl_property (hgprop, GST_PROP_USER_TOTAL_TIME, gtime + ptime);
```

gst_hg_to_graph

Given a hypergraph having a geometric embedding for each of its vertices and edges, construct an ordinary graph containing the individual edges in the embedding. For a rectilinear embedding the parameter `GST_PARAM_GRID_OVERLAY` is used to specify that the edges of the reduced grid graph rather than individual edges of the embedding should be returned.

The original vertices in the hypergraph are marked as *terminals* in the new graph, but the only way[3] to get this information out of the new graph is to print it using function **gst_save_hg()**.

```
gst_hg_ptr gst_hg_to_graph (gst_hg_ptr     H,
                            gst_param_ptr  param,
                            int*           status);
```

`H`	Hypergraph
`param`	Parameter set.
`status`	Status code (zero if the operation was successful and non-zero otherwise).

Returns the new graph which represents the embedding.

Example:

```
/* Assume we are given an embedded hypergraph H */

H2 = gst_hg_to_graph (H, NULL, NULL);

/* Now H2 is a graph of the embedding of H. Print it. */
gst_save_hg (stdout, H2, NULL);
```

[3]In a future release of the library, there will be other means of obtaining this information.

3.10 FST generation and pruning functions

All algorithms for solving geometric Steiner tree problems in GeoSteiner use the two-phase approach that consists of full Steiner tree (FST) generation and concatenation.

FST generation is the process of generating a (hopefully small) set of FSTs that is known to contain a Steiner minimum tree (SMT) as a subset. The input to an FST generation algorithm is the set of terminal points, and the output is an embedded hypergraph in which the vertices correspond to terminals and the edges correspond to FSTs. The embedding of each hyperedge (or FST) is the geometric tree structure of the FST.

In this section we describe the interface to all FST generation algorithms. They are all fairly similar. In addition, a FST *pruning* function is given. This function reduces the set of FSTs — or removes edges from the hypergraph — such that the resulting hypergraph still contains an SMT. This may speed up the following concatenation algorithm, in particular for very large problem instances.

gst_generate_fsts

Given a point set (terminals) in the plane, generate a set of FSTs (hyperedges) known to contain an SMT for the point set. The metric that should be used is passed as a parameter (see section 3.7 for more on creating metric objects). The generated FSTs are returned as edges in an embedded hypergraph.

```
gst_hg_ptr
    gst_generate_fsts (int            nterms,
                       double*        terms,
                       gst_metric_ptr metric,
                       gst_param_ptr  param,
                       int*           status);
```

nterms	Number of terminals.
terms	Terminals in an array of doubles $(x_1, y_1, x_2, y_2, \ldots)$
metric	The metric for which FSTs are to be generated.
param	Parameter set (NULL=default parameters).
status	Status code (zero if successful).

Returns the resulting FSTs in a hypergraph structure.

Example:

```
int            n;
double *       terms;
gst_hg_ptr     hg;
gst_metric_ptr metric;

/* Read points from stdin */
n = gst_get_points (stdin, 0, &terms, NULL);

/* Establish lambda-6 metric */
metric = gst_create_metric (GST_METRIC_UNIFORM, 6, NULL);

/* Generate lambda-6 FSTs */
hg = gst_generate_fsts (n, terms, metric, NULL, NULL);
```

gst_generate_efsts

Given a point set (terminals) in the plane, generate a set of FSTs (hyperedges) known to contain an *Euclidean* SMT for the point set. The FSTs are returned as edges in an embedded hypergraph.

```
gst_hg_ptr
    gst_generate_efsts (int            nterms,
                        double*        terms,
                        gst_param_ptr  param,
                        int*           status);
```

`nterms`	Number of terminals.
`terms`	Terminals in an array of doubles $(x_1, y_1, x_2, y_2, \ldots)$
`param`	Parameter set (`NULL`=default parameters).
`status`	Status code (zero if successful).

Returns the resulting FSTs in a hypergraph structure.

Example:

```
int            n;
double *       terms;
gst_hg_ptr     hg;

/* Read points from stdin */
n = gst_get_points (stdin, 0, &terms, NULL);

/* Generate Euclidean FSTs */
hg = gst_generate_efsts (n, terms, NULL, NULL);
```

gst_generate_rfsts

Given a point set (terminals) in the plane, generate a set of FSTs (hyperedges) known to contain a *rectilinear* SMT for the point set. The FSTs are returned as edges in an embedded hypergraph.

```
gst_hg_ptr
    gst_generate_rfsts (int            nterms,
                        double*        terms,
                        gst_param_ptr  param,
                        int*           status);
```

`nterms`	Number of terminals.
`terms`	Terminals in an array of doubles $(x_1, y_1, x_2, y_2, \ldots)$
`param`	Parameter set (`NULL`=default parameters).
`status`	Status code (zero if successful).

Returns the resulting FSTs in a hypergraph structure.

Example:

```
int            n;
double *       terms;
gst_hg_ptr     hg;

/* Read points from stdin */
n = gst_get_points (stdin, 0, &terms, NULL);

/* Generate rectilinear FSTs */
hg = gst_generate_rfsts (n, terms, NULL, NULL);
```

gst_generate_ofsts

Given a point set (terminals) in the plane, generate a set of FSTs (hyperedges) known to contain an *octilinear* SMT for the point set. The FSTs are returned as edges in an embedded hypergraph.

```
gst_hg_ptr
    gst_generate_ofsts (int            nterms,
                        double*        terms,
                        gst_param_ptr  param,
                        int*           status);
```

`nterms`	Number of terminals.
`terms`	Terminals in an array of doubles $(x_1, y_1, x_2, y_2, \ldots)$
`param`	Parameter set (`NULL`=default parameters).
`status`	Status code (zero if successful).

Returns the resulting FSTs in a hypergraph structure.

Example:

```
int             n;
double *        terms;
gst_hg_ptr      hg;

/* Read points from stdin */
n = gst_get_points (stdin, 0, &terms, NULL);

/* Generate octilinear FSTs */
hg = gst_generate_ofsts (n, terms, NULL, NULL);
```

gst_hg_prune_edges

Given a hypergraph H, return a hypergraph H' that has the same vertices as H, but a (possibly) reduced set of edges such that there still exists an optimal solution to H in H'. The pruning algorithms are metric dependent and require a geometric embedding of the hypergraph vertices and edges.

```
gst_hg_ptr gst_hg_prune_edges (gst_hg_ptr     H,
                               gst_param_ptr  param,
                               int*           status);
```

`H`	Hypergraph.
`param`	Parameter set (`NULL`=default parameters).
`status`	Status code (zero if successful).

Returns new pruned hypergraph.

Example:

```
/* Assume that hg is an FST hypergraph */

gst_hg_ptr hg1;

/* Prune the set of FSTs in hg */

hg1 = gst_hg_prune_edges (hg, NULL, NULL);

/* Hypergraph hg1 now has the same set of vertices as hg,
   but (in most cases) a significantly smaller set of edges that
   still contains an SMT as a subset */
```

3.11 Hypergraph optimization functions

The optimization problem associated with hypergraphs is the *minimum spanning tree (MST) in hypergraph problem*. Solving this problem solves the FST concatenation problem — which is the second of the two phases for solving geometric Steiner tree problems.

The library contains a powerful solver for the general MST in hypergraph problem. This solver uses linear programming and branch-and-cut (or backtrack search for very small problem instances). A large number of parameters can be set to control the solver; consult Appendix A.3, A.4 and A.5 for a complete list of all solver parameters.

A solution state object has type `gst_solver_ptr`. It has an associated hypergraph for which an MST should be found. The solver can be stopped and restarted, e.g., depending on either the quality of (approximate) solutions that are found in the solution process, or on the amount of running time used. The solution state object can contain zero or more feasible (though not necessarily optimal) solutions to the problem. A solution state object refers to both an hypergraph object and a parameter object (from which all necessary parameter values are obtained), as illustrated in Figure 6 on page 16. A demonstration program is given in Figure 5 on page 12.

gst_create_solver

Create a solution state object for a given hypergraph. The solution process is started by calling the function **gst_hg_solve()**, and passing the created object as parameter.

```
gst_solver_ptr
    gst_create_solver (gst_hg_ptr     H,
                       gst_param_ptr  param,
                       int*           status);
```

`H`	Hypergraph.
`param`	Parameter set (`NULL`=default parameters).
`status`	Status code (zero if successful).

Returns new problem solution state object.

An example is given in Section 2.2 (Figure 5 on page 12).

gst_free_solver

Free a solution state object. All memory associated with this solution state object, except from the associated hypergraph and its objects, are destroyed.

```
int gst_free_solver (gst_solver_ptr  solver);
```

`solver`	Solution state object. Does nothing if `NULL`.

Returns zero if the operation was successful and non-zero otherwise.

An example is given in Section 2.2 (Figure 5 on page 12).

gst_hg_solve

Solve a tree problem for a given hypergraph. In the current version, this function by default computes a *minimum spanning tree (MST)* in the hypergraph associated with the given solution state object; depending on the parameters given, this function may also compute an heuristic solution to this problem.

This function can be repeatedly called to solve a (time-consuming) problem, e.g., by setting a CPU time limit for each call. The quality of any solution(s) obtained within the given constraints can be queried by calling **gst_get_solver_status()**.

```
int gst_hg_solve (gst_solver_ptr  solver,
                  int *           reason);
```

`solver`	Solution state object.
`reason`	Reason that the solver exited — see the description below. If this parameter is `NULL`, the reason for exiting is not returned.

The function return value indicates whether any serious errors were encountered in the solution process. If this value is zero it means the solver ran successfully and without problems — although it might have deliberately have been preempted by the user.

A non-zero function return value indicates the error causing the solver to exit prematurely. This could for example be `GST_ERR_BACKTRACK_OVERFLOW` which can happen if one has set the solver to use backtrack search on an instance which is too big for this purpose (`GST_PARAM_SOLVER_ALGORITHM`), i.e., more than 32 hyperedges.

When using default parameters (and when not using abort signals) then a value of zero for the `reason` parameter means that the solution search space was completely exhausted. In this case the optimal solution has been found — unless the problem was found to be infeasible. However, if the user has set any of the solver stopping condition parameters, such as the CPU time limit, the actual reason for exiting the solution process is returned using the `reason` parameter. Possible return values are one of the following:

Macro Name	**Description**
`GST_SOLVE_NORMAL`	Normal exit (search space exhausted)
`GST_SOLVE_GAP_TARGET`	Requested gap target obtained
`GST_SOLVE_LOWER_BOUND_TARGET`	Requested lower bound obtained
`GST_SOLVE_UPPER_BOUND_TARGET`	Requested upper bound obtained
`GST_SOLVE_MAX_BACKTRACKS`	Max. number of backtracks exceeded
`GST_SOLVE_MAX_FEASIBLE_UPDATES`	Max. feasible updates exceeded
`GST_SOLVE_ABORT_SIGNAL`	Abort signal received
`GST_SOLVE_TIME_LIMIT`	CPU time limit exceeded
`GST_SOLVE_BB_STOP_REQUESTED`	Caller requested early termination

An example is given in Section 2.2 (Figure 5 on page 12).

gst_get_solver_status

Return the status of the solution (if any) associated with the given solution state object.

```
int gst_get_solver_status (gst_solver_ptr  solver,
                           int*            status);
```

`solver`	Solution state object.
`status`	Status of the current solution (if any).

Returns zero if the operation was successful and non-zero otherwise.

The value of the `status` parameter is one of the following:

Macro Name	Description
`GST_STATUS_OPTIMAL`	Optimal solution is available
`GST_STATUS_INFEASIBLE`	Problem is infeasible
`GST_STATUS_FEASIBLE`	Search incomplete, feasible solution(s) known
`GST_STATUS_NO_FEASIBLE`	Search incomplete, no feasible solutions known
`GST_STATUS_NO_SOLUTION`	Solver never invoked/hypergraph changed

An example is given in Section 2.2 (Figure 5 on page 12).

gst_get_solver_hypergraph

Return the hypergraph associated with the given solution state object.

```
gst_hg_ptr gst_get_solver_hypergraph (gst_solver_ptr  solver);
```

`solver`	Solution state object.

Pointer to associated hypergraph object.

gst_get_solver_param

Return the hypergraph associated with the given solution state object.

```
gst_param_ptr gst_get_solver_param (gst_solver_ptr  solver);
```

`solver`	Solution state object.

Pointer to associated parameter set object.

gst_hg_solution

Retrieve (one of) the best feasible solutions currently known for a given solution state object.

```
int gst_hg_solution (gst_solver_ptr  solver,
                     int*            nedges,
                     int*            edges,
                     double*         length,
                     int             rank);
```

`solver`	Solution state object.
`nedges`	Number of edges in the returned solution tree.
`edges`	Array of edge numbers in the returned solution tree.
`length`	Length of the returned tree.
`rank`	Rank of the solution that should be returned, where 0 is the best solution (see also discussion below).

Returns zero if the operation was successful and non-zero otherwise.

The maximal number of feasible solutions that will be retained by the solver is determined by the parameter `GST_PARAM_NUM_FEASIBLE_SOLUTIONS`. However, for a given solution state object, the actual number of feasible solutions may be less than this maximum — and even zero.

The function returns `GST_ERR_RANK_OUT_OF_RANGE` when `rank` is less than 0 or greater than or equal to the number of feasible solutions available.

Example:

```
/* We assume that solver is a solution state object.
   This code prints all feasible solutions ordered by their rank. */

int i, rank = 0;
int nedges;
int* edges;
double length;

while (1) {
   /* Get number of edges in this solution.
      Exit when no more solutions are available. */
   if (gst_hg_solution (solver, &nedges, NULL, NULL, rank) != 0)
      break;

   /* Get edge indices and length of solution. */
   edges = (int *) malloc (nedges * sizeof (int));
   gst_hg_solution (solver, NULL, edges, &length, rank);

   /* Print edge indices and length. */
   printf ("Rank %d: Length is %f. Edges:", rank, length);
   for (i = 0; i < nedges; i++)
      printf(" %d", edges[i]);
   printf("\n");
   free (edges);
   rank++;
}
```

gst_get_solver_properties

Return the property list associated with a solution state object.

```
gst_proplist_ptr
    gst_get_solver_properties (gst_solver_ptr  solver);
```

`solver`	Solution state object.

Returns the property list.

Example:

```
/* We assume that solver is defined ...*/
double lower_bound;

if (!gst_get_dbl_property(gst_get_solver_properties(solver),
                          GST_PROP_SOLVER_LOWER_BOUND,
                          &lower_bound) {
   printf("Lower bound for solver object is %f\n", lower_bound);
}
```

3.12 Optimization callback functions

The GeoSteiner API provides the ability to invoke user-written code at key points within the internal GeoSteiner algorithms. This is accomplished by means of callback functions. The user establishes such a callback function by providing two pointers:

- `cb_func`: A pointer to a function to call
- `cb_data`: A pointer to user-specified data to pass to the `cb_func` functions

When `cb_func` is a `NULL` pointer (the default), no callback function is invoked. The `cb_data` pointer can be anything (in particular, a `NULL` pointer — GeoSteiner does nothing with this except pass it as an argument to the callback function.

User-written callback functions are passed the following arguments: The `wherefrom` argument indicates the location within GeoSteiner's internal algorithms from which the callback function has been invoked. The code in the user's callback function should test this parameter to determine whether that particular calling context requires action. There is a single callback function that is invoked from many places within GeoSteiner, but most callback functions are only interested in one (or a small number) of these calling contexts.

The `node` argument is an opaque object that provides access to a variety of internal GeoSteiner data that may be of interest to the callback function. The GeoSteiner API provides various accessor functions that use this `node` object to access these data. Note that the `node` argument is ephemeral — the object it refers to exists *only* during the execution of the callback function. The user must not attempt to use this pointer in any other context (e.g., by storing it in a global variable or in her `cb_data` object or other data structure for later use).

The `cb_data` argument is the data pointer provided by the user when the callback function was established.

The GeoSteiner API defines a macro `GST_BB_CALLBACK_ARGS` that encapsulates the argument list declarations above. Using this macro helps to automatically update the argument lists of user-written callback functions in the event that future versions of GeoSteiner pass additionsl arguments to callback functions.

The `wherefrom` argument can have one of the following values:

- `GST_CALLBACK_BB_LP_SOLVED`: Invoked each time a node finishes solving its LP over the constraint pool. (Solving over the constraint pool is an iterative process that may invoke the LP solver several times until the current LP solution satisfies all constraints in the pool. This callback is only invoked when this iteration completes and all pool constraints are satisfied.)
- `GST_CALLBACK_BB_NEW_UPPER_BOUND`: Invoked whenever an improved upper bound (integer feasible solution) is obtained.
- `GST_CALLBACK_BB_ROOT_NODE_DONE`: Invoked when the root node has finished. For fractional solutions, this is invoked before selecting the branch candidate.

gst_set_bb_callback_func

Set the branch-and-bound callback function (and data) for the given solution state object to the given function and data pointers.

```
int gst_set_bb_callback_func (
        gst_solver_ptr    solver,
        void              (*cb_func) (GST_BB_CALLBACK_ARGS),
        void *            cb_data);
```

Returns status code, which is zero upon success.

Example:

```
void my_callback (int wherefrom, gst_node_ptr node, void * cb_data)
{
printf ("Callback invoked from %d.\n", wherefrom);
}

struct mydata  data;

gst_set_bb_callback_func (solver, my_callback, &data);
```

gst_node_get_solver

A callback node accessor function to get the solution state object on whose behalf this callback function has been invoked.

```
gst_solver_ptr gst_node_get_solver (gst_node_ptr node);
```

Returns solution state object for which the callback function was invoked.

Example:

```
void my_callback (GST_BB_CALLBACK_ARGS)
{
gst_solver_ptr    solver;
  solver = gst_node_get_solver (node);
  ...
}
```

gst_node_get_z

A callback node accessor function to get the LP objective value associated with the given node.

```
double gst_node_get_z (gst_node_ptr node);
```

`node`	The node accessor object passed to the callback function by GeoSteiner.

Returns the LP objective value.

Example:

```
void my_callback (GST_BB_CALLBACK_ARGS)
{
double            z;

  z = gst_node_get_z (node);
  ...
}
```

gst_node_get_lb_status

A callback node accessor function to get the current node's ”lower bound status.”

```
int gst_node_get_lb_status (gst_node_ptr node);
```

Returns an integer having one of the following values:

- `GST_LB_STATUS_INFEASIBLE`
- `GST_LB_STATUS_CUTOFF`
- `GST_LB_STATUS_INTEGRAL`
- `GST_LB_STATUS_FRACTIONAL`
- `GST_LB_STATUS_PREEMPTED`

Example:

```
void my_callback (GST_BB_CALLBACK_ARGS)
{
int status;
  status = gst_node_get_lb_status (node);
  switch (status) {
  case GST_LB_STATUS_INFEASIBLE:   ...   break;
  case GST_LB_STATUS_CUTOFF:       ...   break;
  case GST_LB_STATUS_INTEGRAL:     ...   break;
  case GST_LB_STATUS_FRACTIONAL:   ...   break;
  case GST_LB_STATUS_PREEMPTED:    ...   break;
  default:  abort ();
  }
  ...
}
```

gst_node_get_node_index

A callback node accessor function to get the index of the current node.

```
int gst_node_get_node_index (gst_node_ptr node);
```

Returns the integer node index. Index 0 represents the root node, with child nodes having positive indices. An index of -1 indicates a candidate child node during branch variable selection.

Example:

```
void my_callback (GST_BB_CALLBACK_ARGS)
{
int node_index;
  node_index = gst_node_get_node_index (node);
  printf ("Node %d\n", node_index);
}
```

gst_node_get_parent_node_index

A callback node accessor function to get the index of the parent node of the current node.

```
int gst_node_get_parent_node_index (gst_node_ptr node);
```

Returns the integer index of the parent node. The parent of the root node has index -1.

Example:

```
void my_callback (GST_BB_CALLBACK_ARGS)
{
int parent_node_index;
  parent_node_index = gst_node_get_parent_node_index (node);
  printf ("Parent node %d\n", parent_node_index);
}
```

gst_node_get_node_depth

A callback node accessor function to get the depth of the current node within the branch-and-bound tree. The root node has depth zero.

```
int gst_node_get_node_depth (gst_node_ptr node);
```

Returns the depth of the current node.

Example:

```
void my_callback (GST_BB_CALLBACK_ARGS)
{
int depth;
  depth = gst_node_get_node_depth (node);
  printf ("Node depth %d\n", depth);
}
```

gst_node_get_node_branch_var

A callback node accessor function to get the index of the fractional variable that was branched upon to create the current node from its parent. This is -1 for the root node.

```
int gst_node_get_node_branch_var (gst_node_ptr node);
```

Returns the branch variable used to create the current node.

Example:

```
void my_callback (GST_BB_CALLBACK_ARGS)
{
int bvar;
  bvar = gst_node_get_node_branch_var (node);
  printf ("Branch variable %d\n", bvar);
}
```

gst_node_get_node_branch_direction

A callback node accessor function to get the direction in which the branch variable was constrained to create the current node from its parent. A value of 0 indicates the var=0 branch. A value of 1 indicates the var=1 branch. This is 0 for the root node.

```
int gst_node_get_node_branch_direction (gst_node_ptr node);
```

Returns the direction of the branch used to create the current node.

Example:

```
void my_callback (GST_BB_CALLBACK_ARGS)
{
int dir;
  dir = gst_node_get_node_branch_direction (node);
  printf ("Branch direction %d\n", dir);
}
```

gst_node_get_lp_index

A callback node accessor function to get the number of LPs solved at the current node. (The separation algorithms are run on the solution of each such LP. If violated constraints are found, they are added to the LP which is then re-solved, which increments this index.)

```
int gst_node_get_lp_index (gst_node_ptr node);
```

Returns the index of the most recently solved LP at the current node. The first LP solved receives an index of zero.

Example:

```
void my_callback (GST_BB_CALLBACK_ARGS)
{
int lp_index;
  lp_index = gst_node_get_lp_index (node);
  printf ("LP %d\n", lp_index);
}
```

gst_node_get_solution

A callback node accessor function to get the LP solution vector associated with the given node.

```
int gst_node_get_solution (gst_node_ptr node, double * x);
```

`node`	The node accessor object passed to the callback function by GeoSteiner.
`x`	Address of an array of `double` having sufficiently many elements to receive the solution vector.

Returns an error code, which is zero upon success.

Example:

```
void my_callback (GST_BB_CALLBACK_ARGS)
{
int             nedges;
gst_solver_ptr  solver;
gst_hg_ptr      H;
double          x;
  /* Query number of edges in problem. */
  solver = gst_node_get_solver (node);
  H = gst_get_solver_hypergraph (solver);
  gst_get_hg_edges (H, &nedges, NULL, NULL, NULL);
  /* Allocate array for solution vector. */
  x = malloc (nedges * sizeof (double));

  gst_node_get_solution (node, x);
  ...
  free (x);
}
```

gst_node_get_lb

A callback node accessor function to get the current node's vector of lower bounds.

```
int gst_node_get_lb (gst_node_ptr node, double * lb);
```

`node`	The node accessor object passed to the callback function by GeoSteiner.
`lb`	Address of an array of `double` having sufficiently many elements to receive the lower bound vector.

Returns an error code, which is zero upon success.

Example:

```
void my_callback (GST_BB_CALLBACK_ARGS)
{
int               nedges;
gst_solver_ptr    solver;
gst_hg_ptr        H;
double            lb;
  /* Query number of edges in problem. */
  solver = gst_node_get_solver (node);
  H = gst_get_solver_hypergraph (solver);
  gst_get_hg_edges (H, &nedges, NULL, NULL, NULL);
  /* Allocate array for lower bound vector. */
  lb = malloc (nedges * sizeof (double));

  gst_node_get_lb (node, lb);
  ...
  free (lb);
}
```

gst_node_get_ub

A callback node accessor function to get the current node's vector of upper bounds.

```
int gst_node_get_ub (gst_node_ptr node, double * ub);
```

`node`	The node accessor object passed to the callback function by GeoSteiner.
`ub`	Address of an array of `double` having sufficiently many elements to receive the upper bound vector.

Returns an error code, which is zero upon success.

Example:

```
void my_callback (GST_BB_CALLBACK_ARGS)
{
int                nedges;
gst_solver_ptr     solver;
gst_hg_ptr         H;
double             ub;
  /* Query number of edges in problem. */
  solver = gst_node_get_solver (node);
  H = gst_get_solver_hypergraph (solver);
  gst_get_hg_edges (H, &nedges, NULL, NULL, NULL);
  /* Allocate array for upper bound vector. */
  ub = malloc (nedges * sizeof (double));

  gst_node_get_ub (node, ub);
  ...
  free (ub);
}
```

3.13 Message handling functions

All output messages from GeoSteiner are passed through user-controllable channels. A given channel may write its output to more than one output (screen/files). Channels have type `gst_channel_ptr`.

In this section we describe the functions for creating and freeing channels, for adding output (screen/files) to a channel, and the basic functions for writing to channels.

gst_create_channel

Create a channel with an optional set of options. By default, output is unformatted. In the current version, the only formatted output is Postscript; see function **gst_channel_setopts()** for an example of how to activate Postscript formatting. Consult `geosteiner.h` for the detailed structure of `gst_channel_options`.

```
gst_channel_ptr
    gst_create_channel
        (const gst_channel_options*  chanopts,
         int*                        status);
```

`chanopts`	Channel options (if `NULL` then default options are used).
`status`	Status code (zero if successful).

Returns the new channel object.

Example:

```
/* Create a channel with default options.
   Ignore returned status. */
gst_channel_ptr chan;
chan = gst_create_channel(NULL, NULL);
```

gst_free_channel

Free a channel and all its destinations.

```
int gst_free_channel (gst_channel_ptr  chan);
```

`chan`	Channel object. Does nothing if `NULL`.

Returns zero if the operation was successful and non-zero otherwise.

Example:

```
/* Assume that chan is an existing channel object */
gst_free_channel (chan);

/* All memory used by chan is now freed */
```

gst_channel_getopts

Get channel options.

```
int gst_channel_getopts
        (gst_channel_ptr       chan,
         gst_channel_options*  options);
```

`chan`	Channel opbject.
`options`	Pointer to the channel option structure where channel options should be returned.

Returns zero if the operation was successful and non-zero otherwise.

Example:

```
/* Assume that chan is a channel */
gst_channel_options chanopts;

/* Get options and active Postscript output */
gst_channel_getopts (chan, &chanopts);
chanopts.flags |= GST_CHFLG_POSTSCRIPT;
gst_channel_setopts (chan, &chanopts);
```

gst_channel_setopts

Set channel options.

```
int gst_channel_setopts
        (gst_channel_ptr             chan,
         const gst_channel_options*  options);
```

`chan`	Channel opbject.
`options`	Pointer to the channel option structure that contains new channel options.

Returns zero if the operation was successful and non-zero otherwise.

Example:

```
/* Assume that chan is a channel */
gst_channel_options chanopts;

/* Get options and active Postscript output */
gst_channel_getopts (chan, &chanopts);
chanopts.flags |= GST_CHFLG_POSTSCRIPT;
gst_channel_setopts (chan, &chanopts);
```

gst_channel_add_file

Add a file destination to a channel.

```
gst_dest_ptr
    gst_channel_add_file (gst_channel_ptr  chan,
                          FILE*            fp,
                          int*             status);
```

`chan`	Channel object.
`fp`	File handle.
`status`	Status code (zero if successful).

Returns the new destination object (of type `gst_dest_ptr`).

Example:

```
/* Setup a channel for stdout */
gst_channel_ptr chan;

chan = gst_create_channel (NULL, NULL);
gst_channel_add_file (chan, stdout, NULL);
```

gst_channel_add_functor

Add a function as destination to a channel.

```
typedef size_t
    gst_channel_func (const char*  buf,
                      size_t       cnt,
                      void*        handle);
gst_dest_ptr
    gst_channel_add_functor
                     (gst_channel_ptr    chan,
                      gst_channel_func*  func,
                      void*              handle,
                      int*               status);
```

`chan`	Channel object.
`func`	Function that should be added as destination.
`handle`	Handle used for passing error codes from the function back to the application.
`status`	Status code (zero if successful).

Returns the new destination object (of type `gst_dest_ptr`).

Example:

```
static void
 output_text_to_GUI (void *       handle,
                     const char * text,
                     size_t       nbytes)
{
Widget *            widget = handle;
   my_gui_write_text_to_text_widget (widget, text, nbytes);
}

int main (int argc, char **argv)
{
int                status;
Widget *           widget = my_gui_create_text_widget ();
gst_channel_ptr    mychan = gst_create_channel (NULL, NULL);
gst_param_ptr      myparm = gst_create_param (NULL);

    /* Add functor to write output to GUI window. */
    gst_channel_add_functor (mychan,
                             output_text_to_GUI,
                             widget,
                             &status);
    gst_set_cnh_param (myparm,
                       GST_PARAM_PRINT_SOLVE_TRACE,
                       mychan);
    /* Problems solved using myparm will send */
    /* trace output to the GUI window. */
}
```

gst_channel_rmdest

Remove a destination from a channel.

```
int gst_channel_rmdest (gst_dest_ptr  dest);
```

`dest`	Destination that should be removed.

Returns zero if the operation was successful and non-zero otherwise.

Example:

```
/* Assume that dest is a destination object */
gst_channel_rmdest (dest);

/* Destination object dest is now removed from its channel */
```

gst_channel_write

Write a string to all destinations in a channel.

```
int gst_channel_write (gst_channel_ptr  chan,
                       const char*      text,
                       size_t           nbytes);
```

`chan`	Channel object.
`text`	Buffer with text that should be written.
`nbytes`	Number of bytes in buffer.

Returns zero if the operation was successful and non-zero otherwise.

Example:

```
/* Assume that chan is a channel. */
char* hello = "Hello, World!\n";
gst_channel_write (chan, hello, strlen(hello));
```

gst_channel_printf

Print a formatted string to all destinations in a channel.

```
int gst_channel_printf (gst_channel_ptr  chan,
                        const char*      format,
                        ...) _GST_PRINTF_ARGS (2,3);
```

`chan`	Channel object.
`format`	Printf formatting string.
`...`	Arguments for formatting string.

Returns zero if the operation was successful and non-zero otherwise.

Example:

```
/* Let chan be a channel, and let i1 and i2
   be two integer variables. */
gst_channel_printf (chan, "i1 = %d  i2 = %d\n", i1, i2);
```

3.14 Input and output functions

A number of functions are provided for input and output of hypergraphs. The input/output format can be chosen using parameters. Scaling information can be associated with input points, and numbers can be printed in unscaled using this information.

gst_create_scale_info

Create a scaling information object.

```
gst_scale_info_ptr gst_create_scale_info (int* status);
```

`status`	Status code (zero if successful).

Returns the new scaling information object.

Example:

```
/* Create a new scaling information object
   and use it to hold scaling information for
   a set of points read from stdin. */
int n;
double* terms;
gst_scale_info_ptr scinfo;

scinfo = gst_create_scale_info (NULL);
n = gst_get_points (stdin, 0, &terms, scinfo);
```

gst_free_scale_info

Free a scaling information object.

```
int gst_free_scale_info (gst_scale_info_ptr scinfo);
```

`scinfo`	Scaling information object that should be freed.

Returns zero if the operation was successful and non-zero otherwise.

Example:

```
/* Assume that scinfo is a scaling information object */
gst_free_scale_info (scinfo);

/* All memory used by scinfo is now freed */
```

gst_get_points

Reads a point set from a file (e.g., stdin). Point coordinates should be separated by whitespace. Reads until end-of-file or until a specified number of points have been read.

A scaling information object can be associated with the set of points that are read; if such an object is passed as an argument, this function attempts to find an appropriate scaling for the points to maximize the accuracy of the internal (double) representation. If the scaling information object is `NULL`, no scaling is performed.

```
int gst_get_points (FILE*              fp,
                    int                maxpoints,
                    double**           points,
                    gst_scale_info_ptr scinfo);
```

`fp`	Input file to read from.
`maxpoints`	Maximum number of points to read (if zero then read until end-of-file).
`points`	Array containing read points (which must be allocated by the user *except* when `maxpoints = 0`).
`scinfo`	Scaling information object.

Returns the number of read points.

Example:

```
/* Read a set of points from stdin (until end-of-file).
   A scaling information object is used. */
int n;
double* terms;
gst_scale_info_ptr scinfo;

scinfo = gst_create_scale_info (NULL);
n = gst_get_points (stdin, 0, &terms, scinfo);
```

gst_compute_scale_info_digits

Set up various parameters needed for outputting scaled coordinates. Coordinates/distances are printed with the minimum fixed precision whenever this gives the exact result, that is, if all terminal coordinates are integral, they should always be written without a decimal point. Otherwise we will print the coordinates/distances with full precision.

```
int gst_compute_scale_info_digits
        (int                 nterms,
         double*             terms,
         gst_scale_info_ptr  scinfo);
```

`nterms`	Number of terminals.
`terms`	Terminals in an array of doubles $(x_1, y_1, x_2, y_2, \ldots)$
`scinfo`	Scaling information object that should be modified.

Returns zero if operation was successful and non-zero otherwise.

Example:

```
/* Assume that terms holds a set of n terminals
   and that scinfo is an associated scaling
   information object. Find the minimum number of digits
   necessary when printing unscaled coordinates. */
gst_compute_scale_info_digits (n, terms, scinfo);
```

gst_unscale_to_string

Convert a given internal scaled coordinate to a printable unscaled ASCII string. The internal form is in most cases an integer (to eliminate numeric problems), but the unscaled data may involve decimal fractions.

```
char* gst_unscale_to_string
          (char*              buffer,
           double             val,
           gst_scale_info_ptr  scinfo);
```

`buffer`	Write unscaled string to this buffer. It should be allocated to hold at least 32 characters.
`val`	Double value that should be unscaled.
`scinfo`	Scaling information object.

Returns a pointer to a string holding the unscaled value.

Example:

```
/* Print a set of n terminals in array terms
   to channel chan. Scaling information is
   given by scinfo. */
int i;
char buf1[32], buf2[32];

for (i = 0; i < n; i++) {
   gst_unscale_to_string (buf1, terms[2*i],   scinfo);
   gst_unscale_to_string (buf2, terms[2*i+1], scinfo);
   gst_channel_printf (chan, "(%s, %s)\n", buf1, buf2);
}
```

gst_unscale_to_double

Convert a given internal form coordinate to an unscaled double.

```
double gst_unscale_to_double
            (double              val,
             gst_scale_info_ptr  scinfo);
```

`val`	Double value that should be unscaled.
`scinfo`	Scaling information object.

Returns an unscaled double approximation.

Example:

```
/* Compute an unscaled array of terminal coordinates
   from a scaled set of n terminals in array terms.
   Scaling information is given by scinfo. */
int i;
double* unscaled_terms;

unscaled_terms = (double *) malloc (2 * n * sizeof (double));

for (i = 0; i < 2*n; i++) {
   unscaled_terms[i] = gst_unscale_to_double (terms[i],
                                              scinfo);
}
```

gst_load_hg

Load a hypergraph from an input file. The function creates a new hypergraph and adds the vertices and edges read from the input file. The file format must be one of the FST data formats given in Appendix E.

```
gst_hg_ptr gst_load_hg (FILE*          fp,
                        gst_param_ptr  param,
                        int*           status);
```

`fp`	Input file to read from.
`param`	Parameter set (currently not used).
`status`	Status code (zero if successful).

Returns the hypergraph that is read.

Example:

```
/* Load a hypergraph from stdin */
gst_hg_ptr H;
H = gst_load_hg (stdin, NULL, NULL);
```

gst_save_hg

Print a hypergraph to a file. The print format can be specified by parameter GST_PARAM_SAVE_FORMAT.

```
int gst_save_hg (FILE*          fp,
                 gst_hg_ptr     H,
                 gst_param_ptr  param);
```

fp	Print to this file.
H	Hypergraph that should be printed.
param	Parameter set (NULL=default parameters).

Returns zero if the operation was successful and non-zero otherwise.

Example:

```
/* Print a hypergraph H to stdout using
   the default print format */
gst_save_hg (stdout, H, NULL);
```

3.15 Miscellaneous functions

In this section we describe a few miscellaneous functions, e.g., asynchronous functions that may be used by signal handlers.

gst_deliver_signals

This function is designed to be safely callable from a signal handler. The given signals are delivered to the given solver, which responds to them at some point in the near future. The signals parameter is the bit-wise OR of one or more special signal values defined below.

```
void gst_deliver_signals (gst_solver_ptr  solver,
                          int             gstsignals);
```

`solver`	Solution state object.
`gstsignals`	Bit vector defining the signals that should be delivered to the solver; see table below for a list of possible signals.

Returns nothing.

The following is a list of possible signals that can be delivered to the solver:

Macro Name	**Description**
`GST_SIG_ABORT`	Abort computation
`GST_SIG_FORCE_BRANCH`	Stop cutting and force a branch
`GST_SIG_STOP_TEST_BVAR`	Stop testing branch variables and use the best one seen so far
`GST_SIG_STOP_SEP`	Abort the separation routines and continue with all cuts discovered so far

Example:

```
/* Assume that solver is a solution state object.
   Deliver a signal to force a branch. */
gst_deliver_signals (solver, GST_SIG_FORCE_BRANCH);
```

4 Stand-Alone Programs

Below we first give some examples of program invocations. This is followed by a complete description of each stand-alone program. Note that a short description of each program also can be obtained by running the program with the **-h** option.

The following command will generate a set of 70 random points and compute a rectilinear Steiner minimal tree for it:

```
rand_points 70 | rfst | bb
```

The following computes an Euclidean Steiner minimal tree

```
rand_points 70 | efst | bb
```

and the following computes an octilinear Steiner minimal tree for the same set of points

```
rand_points 70 | ufst | bb
```

Note that rand_points always generates the same sequence of points unless given the **-r** or **-s** option.

The following (Bourne shell) examples can be used to generate complete printable postscript plots for these problem instances:

```
(cat prelude.ps; rand_points 70 | rfst | bb) >rsmt70.ps
(cat prelude.ps; rand_points 70 | efst | bb) >esmt70.ps
(cat prelude.ps; rand_points 70 | ufst | bb) >usmt70.ps
```

The complete set of FSTs can also be plotted as follows:

```
(cat prelude.ps; rand_points 70 | rfst | plotfst -fgo) >rfsts.ps
(cat prelude.ps; rand_points 70 | efst | plotfst -fgo) >efsts.ps
(cat prelude.ps; rand_points 70 | ufst | plotfst -fgo) >ufsts.ps
```

A reduced Hanan grid in the OR-library format (for the rectilinear problem) can be generated as follows:

```
rand_points 70 | rfst | fst2graph
```

By pruning the set of FSTs, an even more reduced grid graph can be generated:

```
rand_points 70 | rfst | prunefst | fst2graph
```

An Euclidean Steiner minimal tree for the `berlin52.tsp` instance from TSPLIB can be constructed and displayed as follows (assuming that the file `berlin52.tsp` is present in your GeoSteiner directory):

```
(cat prelude.ps; lib_points <berlin52.tsp | efst | bb) | gv -
```

rand_points

Generates random point sets. There is considerable flexibility in choosing the size, precision and scaling factor for the generated point coordinates. By default, the coordinates are almost always real numbers, uniformly distributed in the interval $[0, 1)$ (see below for exceptions to this rule). Several pseudo-random generator algorithms are supported. The number of digits per coordinate (both default and maximum) vary by generator, as described below. The following options are permitted:

-b	Binary mode. Generates coordinates that are uniformly distributed doubles in $[0, 1)$, outputting them with full precision.
-d N	Generate decimal numbers having N digits. (See below for default and maximum values, which vary by generator.)
-g G	Use pseudo-random number generator G. (See below.)
-k KEY	Modify default generator seed with KEY, which can be arbitrary text.
-p N	Make N of the coordinate digits be fractional (i.e., to the right of the decimal point). Default is for all digits to be fractional.
-r	Randomize. Use an initial seed chosen from the current date and time.
-s FILE	If FILE exists, read the generator and its initial seed from this file. When finished, write the generator and final seed to this file.

The **-g G** argument allows choosing between the following pseudo-random number generators:

G	**Default : Max Digits**	**Description**
0	4 : 5	The “legacy” random generator. It is based on the original PDP-11 Unix rand(3) function (with all of its ugly warts intact). Its randomness is quite poor.
1	7 : 9	The “new” random generator. It uses a 64-bit shift register with XOR feedback, and produces a reasonable level of randomness.
2	7 : 19	The “AES-256” random generator. It uses the AES-256 block cipher as its fundamental entropy source, producing truly excellent randomness. GeoSteiner must be built with GMP in order for this generator to be available.

If no generator is specified, rand_points will use generator 2 (AES-256), if available. Otherwise, generator 1 is used. The default generator can be overridden using the RAND_POINTS_DEFAULT_GENERATOR environment variable.

Note that using -s to load an intial seed from an existing seed file has the effect of specifying the generator, since there is a data field within the seed file that specifies the generator. (This data field is necessary because the format of the seed file state information is different for each generator.) If both -g and -s are specified, then either (1) the seed file must **not** yet exist, or (2) the generator specified with -g must match that specified within the seed file.

Previous versions of rand_points only supported generators 0 and 1, with the default being to use generator 0 to generate 4-digit integer coordinates. As a special case, when using generator 0 (legacy) with neither the -d nor -p arguments, the current version of rand_points also generates 4-digit integers, thereby replicating the behavior of previous versions of rand_points. Users who prefer the point sets produced by previous versions of rand_points can obtain these same, familiar point sets without any additional command line arguments simply by setting the environment variable

RAND_POINTS_DEFAULT_GENERATOR=0

lib_points

Reads a point set in either TSPLIB or OR-library format from stdin and converts the input to point coordinates as required by **efst**, **rfst** or **ufst**. The program automatically determines the input file type. The program has one optional parameter (which has value 1 by default) that specifies which instance number should be extracted from an OR-library file.

efst

Reads a point set from stdin, and generates a set of Euclidean FSTs that contains at least one Euclidean Steiner minimal tree. The following options are permitted:

-d txt	Description of problem instance.
-g	Use greedy heuristic instead of Smith-Lee-Liebman (more time consuming but generates fewer eq-points).
-k K	Generate only FSTs having at most K terminals. This can save considerable time but can also eliminate FSTs that must be in the optimal Steiner tree (i.e., solutions can become suboptimal).
-m M	Use multiple precision. Larger M use it more. Default is M=0 which disables multiple precision. The use of this option requires that GeoSteiner be configured to use the GNU Multi-Precision arithmetic library (GMP). (See the INSTALL file for more details).
-t	Print detailed timings to stderr.
-v N	Generate the output in version N of the FST data format. Supported versions are 0, 1, 2 and 3. Version 3 is the default.
-Z P V	Set parameter P to value V, e.g. `-ZEPS_MULT_FACTOR 64` sets the epsilon multiplication factor to 64 (`GST_PARAM_EPS_MULT_FACTOR = 64`).

rfst

Reads a point set from stdin, and generates a set of rectilinear FSTs that contains at least one rectilinear Steiner minimal tree. The following options are permitted:

-d txt	Description of problem instance.
-k K	Generate only FSTs having at most K terminals. This can save time but can also eliminate FSTs that must be in the optimal Steiner tree (i.e., solutions can become suboptimal).
-t	Print detailed timings to stderr.
-v N	Generate the output in version N of the FST data format. Supported versions are 0, 1, 2 and 3. Version 3 is the default.
-Z P V	Set parameter P to value V, e.g. `-ZINCLUDE_CORNERS 0` disables the generation of corner points (`GST_PARAM_INCLUDE_CORNERS = 0`)

ufst

Reads a point set from stdin, and generates a set of uniformly-oriented FSTs that contains at least one uniformly-oriented Steiner minimal tree. The following options are permitted:

-d txt	Description of problem instance.
-k K	Generate only FSTs having at most K terminals. This can save time but can also eliminate FSTs that must be in the optimal Steiner tree (i.e., solutions can become suboptimal).
-l L	Number of orientations (default: 4).
-t	Print detailed timings to stderr.
-v N	Generate the output in version N of the FST data format. Supported versions are 0, 1, 2 and 3. Version 3 is the default.
-Z P V	Set parameter P to value V, e.g. `-ZINCLUDE_CORNERS 0` disables the generation of corner points (`GST_PARAM_INCLUDE_CORNERS = 0`)

bb

The FST concatenation algorithm using branch-and-cut to solve an IP formulation of the problem. The FST data is read from stdin and a plot of the solution is produced on stdout in an “incomplete” postscript form. A printable postscript file can be obtained by *prepending* the file ”prelude.ps” to the program output. If you want this file to be included in some other document then it needs a bounding box. This can be obtained by running it through **eps2eps** (GhostScript 6.01 or later).

Various trace messages appear in the output as postscript comments. (The name **bb** is for branch-and-bound – note that the name **bc** is already taken on Unix.) The following options are permitted:

-2	Omit all 2-terminal Subtour Elimination Constraints (SEC’s) from the initial constraint pool.
-b	Disable ”strong branching”, which chooses branching variables very carefully.
-B N	Set branch variable selection policy. N=0: naive max of mins, N=1: smarter lexicographic max of mins (default), N=2: product of improvements.
-c P	Pathname of checkpoint file to restore (if present) and/or update. The files are actually named P.chk and P.ub, with temporary files named P.tmp, P.new and P.nub.
-f	The only information dumped is the FSTs in the best solution found. This can then be given to dumpfst/plotfst. E.g. `rand_points | efst | bb -f | dumpfst -sl`
-H	Force the use of the backtrack search. This will result in an error if there are more than 32 edges. Note that there is still a limit on the number of backtracks (`GST_PARAM_MAX_BACKTRACKS`). If using this option one might also want to set backtrack limit to infinity (otherwise an optimal solution might not be found).
-l T	Sets a CPU time limit (in seconds) of T. Example CPU times are: `-l 3days2hours30minutes15seconds`, `-l 1000seconds`, `-l 1000` and `-l 2h30m`.
-m P	Merge constraints from checkpoint file P with those of the formulation.

-n N	Output N best solutions (default: 1).
-r	Plot the optimal LP relaxation solution for the root node, but only if it is fractional.
-R	When optimal root LP relaxation is obtained, determine for each LP iteration the number of final constraints whose first violation occurred during that iteration.
-t	Do *not* include the title string in the postscript output (name, length and time).
-T N	Search N times more thoroughly for strong branching variables.
-u B	Specify B to be the initial upper bound assumed by the branch-and-bound.
-z N	Set the target number of pool non-zeros to N.
-Z P V	Set parameter P to value V, e.g. `-ZGAP_TARGET 0.5` sets `GST_PARAM_GAP_TARGET = 0.5`.

When configured to use CPLEX, the following additional option is permitted:

-a M N	Force CPLEX allocation to be at least M rows and N non-zeros.

When configured to use `lp_solve`, the following additional options are permitted:

-p	Turn on the use of perturbations. This is the method that `lp_solve_2.3` uses to deal with degenerate problems.
-s	Turn on the use of problem scaling. Once again a rather crude attempt to address problems that are badly behaved numerically.

The following “grep-able” items appear in the output within postscript comments, and may be potentially useful:

`@0` The instance description from the FST data file.

`@1` Summary statistics:

- Number of terminals
- Number of FSTs/hyperedges

  - Number of branch-and-bound nodes
  - Number of LPs solved
  - Phase 1 CPU time (FST generation)
  - Phase 2 CPU time (branch-and-cut)
  - Total CPU time

`@2` LP/IP statistics:

  - Length of optimal Steiner tree
  - Length of LP relaxation at root node
  - Percent of LP/IP ”gap”
  - # of LPs solved for root node
  - CPU time for root node
  - Percent reduction of SMT over MST

`@3` Initial constraint pool statistics:

  - Number of rows in pool
  - Number of non-zeros in pool
  - Number of rows in LP tableau
  - Number of non-zeros in LP tableau

`@4` Constraint pool statistics for root node:

  - Number of rows in pool
  - Number of non-zeros in pool
  - Number of rows in LP tableau
  - Number of non-zeros in LP tableau

`@5` Final constraint pool statistics:

  - Number of rows in pool
  - Number of non-zeros in pool
  - Number of rows in LP tableau

- Number of non-zeros in LP tableau

`@6` Statistics on FSTs occurring in the SMT:

- Number of FSTs in SMT
- Average FST size in SMT
- Maximum FST size in SMT
- Number of FSTs of size 2 in SMT
- Number of FSTs of size 3 in SMT
- Number of FSTs of size 4 in SMT
- Number of FSTs of size 5 in SMT
- Number of FSTs of size 6 in SMT
- Number of FSTs of size 7 in SMT
- Number of FSTs of size 8 in SMT
- Number of FSTs of size 9 in SMT
- Number of FSTs of size 10 in SMT
- Number of FSTs of size $>$ 10 in SMT

`@C` Coordinates of a Steiner point in the optimal solution. The Steiner points form a ”certificate” of the optimal solution since the optimal solution can be reconstructed by computing a minimum spanning tree of the original terminals and these Steiner points.

`@D` Deletion of slack rows from LP tableau.

`@LO` / `@LN` This pair of messages is emitted every time the lower bound gets tighter. They contain the CPU time and the old/new bound, as well as the old/new gap percentages. These can be plotted (i.e., using gnuplot) to graphically show the convergence rate of the algorithm.

`@NC` Creation of a new branch-and-bound node:

- Node number
- Parent node number
- Branch variable

- Branch direction
- Objective value (the real LP objective is at least this value)

`@PAP`	Adding ”pending” pool constraints to the LP tableau.
`@PL`	State of LP tableau constraints.
`@PMEM`	Constraint pool memory status. Printed before and after each garbage collection, and after adding new/initial constraints to the pool.
`@r`	Experimental output from -R switch.
`@RC`	Experimental output from -R switch.
`@UO` / `@UN`	This pair of messages is emitted every time the upper bound gets tighter. They contain the CPU time and the old/new bound, as well as the old/new gap percentages. These can be plotted (i.e., using gnuplot) to graphically show the convergence rate of the algorithm.

prunefst

Reduce the set of FSTs generated by **efst**, **rfst** or **ufst** while still retaining at least one optimal solution among the remaining set of FSTs. This program can reduce the time to solve the FST concatenation problem considerably, but is only worthwhile for large instances. The following options are permitted:

-b	Use linear space and logarithmic time lookup for BSDs.
-d txt	Description of problem instance.
-t	Print detailed timings to stderr.
-v N	Generate the output in version N of the FST data format. Supported versions are 0, 1, 2 and 3. Version 3 is the default.
-Z P V	Set parameter P to value V, e.g. `-ZEPS_MULT_FACTOR 64` sets the epsilon multiplication factor to 64 (`GST_PARAM_EPS_MULT_FACTOR = 64`).

dumpfst

Dumps readable information about generated FSTs. There are two forms of this command, each producing a different type of output. The first form of the command is obtained whenever the **-d** or **-h** switches are used. These switches provide summary information *only* — FST statistics, and/or a histogram of FST sizes:.

-d	Display statistics about FSTs.
-h	Display histogram of FST sizes.
-a	Include all FSTs in histogram, even those that were "pruned" by the FST generator or a pruning algorithm.

The second form of the command is obtained when neither **-d** nor **-h** are specified. This form dumps all of the FSTs in a readable form. Each line of output represents a single FST, listing its terminal numbers (0 through N-1). The terminals are listed in the same order that they occur in the actual data structures, although they can optionally be sorted in numeric order. The length of each FST can optionally be appended to each line:

-l	Append the FST length to each output line.
-s	Terminals of each FST are listed in numeric (sorted) order instead of internal order.
-a	Include all FSTs, even those that were "pruned" by the FST generator or a pruning algorithm.

plotfst

Program to generate various plots of FSTs in an FST data file. Reads the FST data file on stdin and produces postscript on stdout for the plots indicated by the command line switches:

-f	Prints all FSTs, 12 FSTs per page.
-g	Prints FSTs in “grouped” fashion, 12 groups per page.
-o	Prints all FSTs overlaid together.
-p	Prints only the points, no FSTs.

Note that the file `prelude.ps` must be *prepended* to the output of this program in order to have a complete postscript document.

fst2graph

Reads FSTs from stdin and produces an (ordinary) graph on stdout representing the FSTs. For the rectilinear problem, the FSTs are overlaid on the Hanan grid and the remaining Hanan grid is output. For the Euclidean problem the set of all terminals and Steiner points in all FSTs forms the set of vertices and the line segments form the edges. Output data is printed in the OR-library format by default, but the SteinLib format is also supported:

-b N	For version 4 output (STEINLIB_INT), generate integer edge weights as unsigned N-bit integers (default is N=64).
-d txt	Description of problem instance.
-e	Generate the edge graph for the rectilinear problem.
-u	Output unscaled (fractional) data for the rectilinear problem.
-v N	Generate version N format output.

References

[1] A. B. Kahng and G. Robins. A New Class of Iterative Steiner Tree Heuristics with Good Performance. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 11(7):893–902, 1992.

[2] B. K. Nielsen, P. Winter, and M. Zachariasen. An Exact Algorithm for the Uniformly-Oriented Steiner Tree Problem. In *Proceedings of the 10th European Symposium on Algorithms, Lecture Notes in Computer Science*, volume 2461, pages 760–772. Springer, 2002.

[3] J. S. Salowe and D. M. Warme. Thirty-Five-Point Rectilinear Steiner Minimal Trees in a Day. *Networks*, 25(2):69–87, 1995.

[4] J. M. Smith, D. T. Lee, and J. S. Liebman. An $O(n \log n)$ Heuristic for Steiner Minimal Tree Problems on the Euclidean Metric. *Networks*, 11:23–29, 1981.

[5] D. M. Warme. *Spanning Trees in Hypergraphs with Applications to Steiner Trees*. Ph.D. thesis, Computer Science Dept., The University of Virginia, 1998.

[6] D. M. Warme, P. Winter, and M. Zachariasen. Exact Algorithms for Plane Steiner Tree Problems: A Computational Study. In D.-Z. Du, J. M. Smith, and J. H. Rubinstein, editors, *Advances in Steiner Trees*, pages 81–116. Kluwer Academic Publishers, Boston, 2000.

[7] P. Winter. An Algorithm for the Steiner Problem in the Euclidean Plane. *Networks*, 15:323–345, 1985.

[8] P. Winter and M. Zachariasen. Euclidean Steiner Minimum Trees: An Improved Exact Algorithm. *Networks*, 30:149–166, 1997.

[9] M. Zachariasen. Rectilinear Full Steiner Tree Generation. *Networks*, 33:125–143, 1999.

[10] M. Zachariasen and P. Winter. Concatenation-Based Greedy Heuristics for the Euclidean Steiner Tree Problem. *Algorithmica*, 25:418–437, 1999.

A Library Parameters

The parameters in GeoSteiner are divided into five groups: FST generation parameters (Section A.1), LP solver parameters (Section A.2), hypergraph solver algorithmic options (Section A.3), hypergraph solver stopping conditions (Section A.4), and hypergraph solver input/output options (Section A.5).

Parameters are modified as described in Section 3.6. Each parameter is uniquely identified by its macro name beginning with (`GST_PARAM`). Below the effect of each parameter is described. Also, the type of each parameter (`double`, `int`, `char*` or `gst_channel_ptr`) and range of possible values are given.

A.1 FST generation parameters

GST_PARAM_MAX_FST_SIZE `int`

Maximum size (number of terminals spanned) of generated FSTs.

Values
Any number greater than or equal to 2 (default: `INT_MAX`).

GST_PARAM_INCLUDE_CORNERS `int`

Include corners of bent edges in FSTs in hypergraph embedding. Applies to rectilinear and uniform-orientation metric FST generators. Including corners makes the embedding easier to draw.

Values

GST_PVAL_INCLUDE_CORNERS_DISABLE	0	(default)
GST_PVAL_INCLUDE_CORNERS_ENABLE	1	

GST_PARAM_EFST_HEURISTIC `int`

Heuristic used in the Euclidean FST generator: Smith-Lee-Liebman or Zachariasen-Winter. The latter is slower but prunes more eq-points; it is therefore recommended for large and/or difficult instances.

Values

GST_PVAL_EFST_HEURISTIC_SLL	0	(default)
GST_PVAL_EFST_HEURISTIC_ZW	1	

GST_PARAM_EPS_MULT_FACTOR `int`

Epsilon multiplication factor F used in floating point comparisons. The maximum relative error is expected to be at most F × `DBL_ESPILON`.

Values
Any number greater than or equal to 1 (default: 32).

`GST_PARAM_MULTIPLE_PRECISION` `int`

Use GNU Multi-Precision arithmetic library (GMP) in the Euclidean FST generator in order to improve numerical precision of computed eq-points: 0: off; 1: use GMP with 1 Newton iteration; 2: use GMP with 1 or more Newton iterations, stopping when a convergence test indicates that 1/2 ULP of precision has been obtained.

Values

`GST_PVAL_MULTIPLE_PRECISION_OFF`	0	(default)
`GST_PVAL_MULTIPLE_PRECISION_ONE_ITER`	1	
`GST_PVAL_MULTIPLE_PRECISION_MORE_ITER`	2	

`GST_PARAM_INITIAL_EQPOINTS_TERMINAL` `int`

Number of eq-points initially allocated per terminal in the Euclidean FST generator. Although eq-point storage is added dynamically when needed, some large or difficult instances run out of memory if the initial allocation is insufficient.

Values
Any number greater than or equal to 1 (default: 100).

`GST_PARAM_BSD_METHOD` `int`

Data structure for holding bottleneck Steiner distances (BSD). Either quadratic space and *constant* time lookup or linear space and *logarithmic* time lookup. The latter is recommended for very large instances.

Values

`GST_PVAL_BSD_METHOD_CONSTANT`	0	(default)
`GST_PVAL_BSD_METHOD_LOGARITHMIC`	1	

A.2 LP solver parameters

`GST_PARAM_LP_SOLVE_PERTURB` `int`

Use perturbations when solving LPs (only applicable when using lp_solve as LP-solver).

Values

`GST_PVAL_LP_SOLVE_PERTURB_DISABLE`	0	(default)
`GST_PVAL_LP_SOLVE_PERTURB_ENABLE`	1	

`GST_PARAM_LP_SOLVE_SCALE` `int`

Use scaling when solving LPs (only applicable when using lp_solve as LP-solver).

Values

`GST_PVAL_LP_SOLVE_SCALE_DISABLE`	0	(default)
`GST_PVAL_LP_SOLVE_SCALE_ENABLE`	1	

`GST_PARAM_CPLEX_MIN_ROWS` `int`

Force the LP solver allocation to be at least N rows (only applicable when using CPLEX as LP-solver).

Values
Any non-negative number (default: 0).

`GST_PARAM_CPLEX_MIN_NZS` `int`

Force the LP solver allocation to be at least N non-zeros (only applicable when using CPLEX as LP-solver).

Values
Any non-negative number (default: 0).

A.3 Hypergraph solver algorithmic options

`GST_PARAM_SOLVER_ALGORITHM` `int`

Hypergraph solver algorithm: Branch-and-cut, backtrack search, or chosen automatically. Backtrack search is only applicable if the instance has 32 or fewer hyperedges. Also note that some stopping conditions — such as UB/LB gap — are *not* feasible for backtrack search. The automatic algorithm uses backtrack search when the instance is small (see parameters `GST_PARAM_BACKTRACK_MAX_VERTS` and `GST_PARAM_BACKTRACK_MAX_EDGES`); furthermore, it switches to branch-and-cut when the the backtrack limit `GST_PARAM_MAX_BACKTRACKS` is hit.

Values

`GST_PVAL_SOLVER_ALGORITHM_AUTO`	0	(default)
`GST_PVAL_SOLVER_ALGORITHM_BRANCH_AND_CUT`	1	
`GST_PVAL_SOLVER_ALGORITHM_BACKTRACK_SEARCH`	2	

`GST_PARAM_NUM_FEASIBLE_SOLUTIONS` `int`

Number N of stored feasible solutions (top N solutions). A value of N for this parameter instructs the solver to retain the N best feasible solutions discovered.

Values

Any number greater than or equal to 1 (default: 1).

`GST_PARAM_BRANCH_VAR_POLICY` `int`

Branch variable policy. 0: naive max of mins, 1: smarter lexicographic max of mins, 2: product of improvements; 3: weak branching. All policies except the last one use strong branching.

Values

`GST_PVAL_BRANCH_VAR_POLICY_NAIVE`	0

`GST_PVAL_BRANCH_VAR_POLICY_SMART`	1	(default)
`GST_PVAL_BRANCH_VAR_POLICY_PROD`	2	
`GST_PVAL_BRANCH_VAR_POLICY_WEAK`	3	

`GST_PARAM_CHECK_BRANCH_VARS_THOROUGHLY` `int`

Search N times more thoroughly for strong branching variables.

Values
Any number from 1 to 1000 (default: 1).

`GST_PARAM_TARGET_POOL_NON_ZEROS` `int`

Target number of pool non-zeros; target is computed automatically when value is zero.

Values
Any non-negative number (default: 0).

`GST_PARAM_SEED_POOL_WITH_2SECS` `int`

This parameter controls whether or not to seed the initial constraint pool with all 2-terminal Subtour Elimination Constraints (SECs). Most problems have relatively few of these, but some problems (such as those with many edges containing a large number of vertices) can blow up unless this is disabled.

Values

`GST_PVAL_SEED_POOL_WITH_2SECS_DISABLE`	0	
`GST_PVAL_SEED_POOL_WITH_2SECS_ENSABLE`	1	(default)

`GST_PARAM_INITIAL_UPPER_BOUND` `double`

Value of initial upper bound for problem being solved.

Values
Any number (default: `DBL_MAX`).

`GST_PARAM_CHECK_ROOT_CONSTRAINTS` `int`

When the optimal root LP relaxation is obtained, determine for each LP iteration the number of final constraints whose first violation occurred during that iteration. This option creates a temporary file to hold the LP solution vector from each iteration. This file can grow very large.

Values

`GST_PVAL_CHECK_ROOT_CONSTRAINTS_DISABLE`	0	(default)
`GST_PVAL_CHECK_ROOT_CONSTRAINTS_ENABLE`	1	

`GST_PARAM_LOCAL_CUTS_MODE` `int`

Local cuts mode: 0: disable local cuts; 1: apply local cuts only when no subtour violation exists; 2: apply local cuts to congested components that contain no subtour violations; 3: apply local cuts in both cases.

Values

`GST_PVAL_LOCAL_CUTS_MODE_DISABLE`	0	(default)
`GST_PVAL_LOCAL_CUTS_MODE_SUBTOUR_RELAXATION`	1	
`GST_PVAL_LOCAL_CUTS_MODE_SUBTOUR_COMPONENTS`	2	
`GST_PVAL_LOCAL_CUTS_MODE_BOTH`	3	

`GST_PARAM_LOCAL_CUTS_MAX_VERTICES` `int`

Local cuts will not be attempted for any subproblem having more than this number of vertices.

Values
Any number from 0 to 80 (default: 80).

`GST_PARAM_LOCAL_CUTS_MAX_EDGES`

`int`

Local cuts will not be attempted for any subproblem having more than this number of edges.

Values
Any number from 0 to 256 (default: 256).

`GST_PARAM_LOCAL_CUTS_VERTEX_THRESHOLD` `double`

A threshold value α that prohibits local cuts on any fractional component $C = (V', E')$ of a parent problem $H = (V, E)$ unless $|V'| < \alpha * |V|$.

Values
Any number from 0 to 1 (default: 0.75).

`GST_PARAM_LOCAL_CUTS_MAX_DEPTH` `int`

Maximum recursive depth of local cuts. 0: disable local cuts; 1: enable local cuts with no recursion; 2: enable local cuts with two recursive levels. -1: enable local cuts with recursion to any depth.

Values

`GST_PVAL_LOCAL_CUTS_MAX_DEPTH_DISABLE`	0	
`GST_PVAL_LOCAL_CUTS_MAX_DEPTH_ONELEVEL`	1	(default)
`GST_PVAL_LOCAL_CUTS_MAX_DEPTH_TWOLEVELS`	2	
`GST_PVAL_LOCAL_CUTS_MAX_DEPTH_ANYLEVEL`	-1	

`GST_PARAM_LOCAL_CUTS_TRACE_DEPTH` `int`

Tracing of local cuts. 0: do not trace local cuts or their recursive invocations; 1: trace first level of local cuts; 2: trace first two levels of local cuts; -1: trace any level of local cuts.

Values

`GST_PVAL_LOCAL_CUTS_TRACE_DEPTH_DISABLE`	0	(default)
`GST_PVAL_LOCAL_CUTS_TRACE_DEPTH_ONELEVEL`	1	
`GST_PVAL_LOCAL_CUTS_TRACE_DEPTH_TWOLEVELS`	2	
`GST_PVAL_LOCAL_CUTS_TRACE_DEPTH_ANYLEVEL`	-1	

GST_PARAM_MAX_CUTSET_ENUMERATE_COMPS `int`

Controls the behavior of the zero-weight cutset separation algorithm, which looks for multiple connected components. If the number N of connected components does not exceed this threshold, then the separator generates one cutset constraint for each of the $2^N - 2$ possible combinations of connected components (excluding the two combinations that take all or none of the components). This parameter controls an exponential process, so setting it too high can easily swamp the solver with constraints.

Values
Any number from 0 to 11 (default: 5)

GST_PARAM_SEC_ENUM_LIMIT `int`

Congested components having at most this number of vertices are exhaustively searched to find all violated subtour elimination constraints. A component with N vertices has $2^N - N - 1$ possible subtour elimination constraints. This parameter therefore controls an exponential process — setting it too high can easily swamp the solver with constraints or increase runtime.

Values
Any number from 0 to 16 (default: 10)

GST_PARAM_BACKTRACK_MAX_VERTS `int`

Backtrack search should only be attempted for solving MST in hypergraph problem when the number of vertices is smaller than this value.

Values
Any number from 0 to 32 (default: 8).

`GST_PARAM_BACKTRACK_MAX_EDGES` `int`

Backtrack search should only be attempted for solving MST in hypergraph problem when the number of edges is smaller than this value.

Values
Any number from 0 to 32 (default: 12).

`GST_PARAM_MAX_BACKTRACKS` `int`

Maximum number of distinct partial solution nodes to enumerate during a single run of the backtrack search algorithm. Note that if this limit is hit, the solver might exit without having found an optimal solution.

Values
Any non-negative number (default: 10000).

`GST_PARAM_SPARSE_SUBTOURS` `int`

There are two forms for hypergraph subtour constraints: the “classic” subtour, and the form obtained by subtracting the “classic” inequality from the single equation of the formulation. If enabled, then generate the form that has the least number of non-zeros (i.e., more sparse). If disabled, always generate the “classic” subtour inequality.

Values

`GST_PVAL_SPARSE_SUBTOURS_DISABLE`	0	
`GST_PVAL_SPARSE_SUBTOURS_ENABLE`	1	(default)

`GST_PARAM_ZERO_WEIGHT_CUTSETS_METHOD` `int`

Controls the behavior of the zero-weight cutset separation algorithm, which is invoked when the support hypergraph of the current LP solution consists of two or more connected components. For each such connected component, this algorithm generates either: (1) a pair of complementary subtour inequalities; or (2) an actual cutset inequality (*not* recommended).

Values

`GST_PVAL_ZERO_WEIGHT_CUTSETS_METHOD_SUBTOURS`	0	(default)
`GST_PVAL_ZERO_WEIGHT_CUTSETS_METHOD_CUTSET`	1	

`GST_PARAM_STRENGTHEN_REDUCE` `int`

If enabled, attempt to strengthen generated subtour inequalities by performing the various reductions (greedy vertex deletion, connected components and biconnected components) on the subhypergraph induced by the vertices of the subtour.

Values

`GST_PVAL_STRENGTHEN_REDUCE_DISABLE`	0	
`GST_PVAL_STRENGTHEN_REDUCE_ENABLE`	1	(default)

`GST_PARAM_GENERATE_UNSTRENGTHENED` `int`

If enabled, generate violated subtours in their original form — before any strengthening is performed. (This is implcitly enabled if `GST_PARAM_STRENGTHEN_REDUCE` is disabled to prevent all violated subtours from being discarded.)

Values

`GST_PVAL_GENERATE_UNSTRENGTHENED_DISABLE`	0	
`GST_PVAL_GENERATE_UNSTRENGTHENED_ENABLE`	1	(default)

`GST_PARAM_INITIAL_PRIMAL_HEURISTIC` `int`

If enabled, run the primal heuristic at the very start, (before any LPs have been solved). Normally the primal heuristic is given the current LP solution as a “hint” for constructing good solutions, but it can also run no such solution. This option is normally disabled because any such upper bound is usually replaced quickly on the next invocation of the primal heuristic, after the first LP has been solved. Turning this option on permits a very quick solution.

Values

`GST_PVAL_INITIAL_PRIMAL_HEURISTIC_DISABLE`	0	(default)
`GST_PVAL_INITIAL_PRIMAL_HEURISTIC_ENABLE`	1	

A.4 Hypergraph solver stopping conditions

GST_PARAM_CPU_TIME_LIMIT `double`

CPU time limit for solver (in seconds); when the limit is zero, *no* CPU time limit is imposed.

Values
Any non-negative number (default: 0).

GST_PARAM_GAP_TARGET `double`

Exit solver when ratio UB/LB between the upper bound (UB) and the lower bound (LB) is less than or equal to this threshold; e.g., if target is 1.01, the solver stops when a solution within 1% from the optimum has been found.

Values
Any number greater than or equal to 1 (default: 1).

GST_PARAM_UPPER_BOUND_TARGET `double`

Exit solver when a feasible solution whose length is at most this value is found.

Values
Any number (default: `-DBL_MAX`).

GST_PARAM_LOWER_BOUND_TARGET `double`

Exit solver when the lower bound becomes greater than or equal to this value.

Values
Any number (default: `DBL_MAX`).

GST_PARAM_MAX_FEASIBLE_UPDATES

`int`

Exit solver when N feasible solution updates have been made (zero means no limit). A feasible update is either an insertion of a solution of any quality into the (non-full) set of solutions, or a replacement of an inferior solution with an improved solution in the (full) set of solutions. The size of the solution set is specified using parameter `GST_PARAM_NUM_FEASIBLE_SOLUTIONS`.

Values
Any non-negative number (default: 0).

`GST_PARAM_BB_NODE_LIMIT`

`int`

Stop the optimization after processing this many branch-and-bound nodes. If this parameter is zero, there is no limit on the number of branch-and-bound nodes processed.

Values
Any non-negative number (default: 0).

`GST_PARAM_BB_LP_LIMIT`

`int`

Stop the optimization after processing this many LPs (optimize / separate iterations). The number of LPs processed is a global number that completely disregards branch-and-bound node boundaries. If this parameter is zero, there is no limit on the number of LPs processed.

Values
Any non-negative number (default: 0).

`GST_PARAM_INITIAL_PRIMAL_HEUR_STOP`

`int`

If this parameter and INITIAL_PRIMAL_HEURISTIC are both enabled, optimization stops immediately after this initial invocation of the primal heuristic, with a status of GST_SOLVE_BB_STOP_REQUESTED. No LPs are solved, nor is any branch-and-bound performed. This permits a very quick solution, but no lower bound will be available, and therefore no indication of solution quality.

Values

GST_PVAL_INITIAL_PRIMAL_HEUR_STOP_DISABLE	0	(default)
GST_PVAL_INITIAL_PRIMAL_HEUR_STOP_ENABLE	1	

A.5 Hypergraph solver input/output options

`GST_PARAM_SAVE_FORMAT` `int`

Format used by **gst_hg_save()** when saving a hypergraph to a file: 0: OR-library format; 1: SteinLib format; 2: GeoSteiner FST format version 2; 3: GeoSteiner FST format version 3; 4: SteinLib format with integer edge weights.

Values

`GST_PVAL_SAVE_FORMAT_ORLIBRARY`	0	
`GST_PVAL_SAVE_FORMAT_STEINLIB`	1	
`GST_PVAL_SAVE_FORMAT_VERSION2`	2	
`GST_PVAL_SAVE_FORMAT_VERSION3`	3	(default)
`GST_PVAL_SAVE_FORMAT_STEINLIB_INT`	4	

`GST_PARAM_SAVE_INT_NUMBITS` `int`

Number of bits of precision to use in final integer edge weights when using **gst_hg_save()** to save hypergraphs having Euclidean metric problems in "integer" SteinLib format (**GST_PARAM_SAVE_FORMAT** set to **GST_PVAL_SAVE_FORMAT_STEINLIB_INT**).

Values

Value must be at least 32. Default value is 64.

`GST_PARAM_GRID_OVERLAY` `int`

Used by function **gst_hg_to_graph()** to specify that the edges of the reduced grid graph rather than individual edges of the embedding should be returned (only applicable for the rectilinear metric).

Values

`GST_PVAL_GRID_OVERLAY_DISABLE`	0	
`GST_PVAL_GRID_OVERLAY_ENABLE`	1	(default)

`GST_PARAM_DETAILED_TIMINGS_CHANNEL`

`gst_channel_ptr`

Detailed timing is written to this channel.

Values
Any valid channel pointer (default: `NULL`).

`GST_PARAM_PRINT_SOLVE_TRACE` `gst_channel_ptr`

Solver output trace is written to this channel.

Values
Any valid channel pointer (default: `NULL`).

`GST_PARAM_CHECKPOINT_FILENAME` `char*`

Pathname P of checkpoint file to restore (if present) and/or update. The files are actually named P.chk and P.ub, with temporary files named P.tmp, P.new and P.nub.

Values
Any valid pathname (default: `NULL`).

`GST_PARAM_CHECKPOINT_INTERVAL` `double`

Perform checkpointing of solver process at a time interval (in seconds) given by this parameter.

Values
Any number between 0 and 1000000 (default: 3600). A value of 0 means that no checkpointing should be performed.

`GST_PARAM_MERGE_CONSTRAINT_FILES`

`char*`

A colon-separated list of pathnames of checkpoint files. All constraints from the constraint pool of each listed checkpoint file are merged into the solver's constraint pool before solving the current hypergraph problem.

Values
A colon-separated list of pathnames of checkpoint files (default: `NULL`).

B Hypergraph Properties

The following table shows the properties currently accessible in a hypergraph instance. Read more about properties in Section 3.8.

Property	Value
GST_PROP_HG_HALF_FST_COUNT	10000
GST_PROP_HG_GENERATION_TIME	20000
GST_PROP_HG_MST_LENGTH	20001
GST_PROP_HG_PRUNING_TIME	20002
GST_PROP_HG_INTEGRALITY_DELTA	20003
GST_PROP_HG_NAME	30000

C Solver Properties

The following table shows the properties currently accessible in a solver object. Read more about properties in Section 3.8.

Property	Value
GST_PROP_SOLVER_ROOT_OPTIMAL	11000
GST_PROP_SOLVER_ROOT_LPS	11001
GST_PROP_SOLVER_NUM_NODES	11002
GST_PROP_SOLVER_NUM_LPS	11003
GST_PROP_SOLVER_INIT_PROWS	11004
GST_PROP_SOLVER_INIT_PNZ	11005
GST_PROP_SOLVER_INIT_LPROWS	11006
GST_PROP_SOLVER_INIT_LPNZ	11007
GST_PROP_SOLVER_ROOT_PROWS	11008
GST_PROP_SOLVER_ROOT_PNZ	11009
GST_PROP_SOLVER_ROOT_LPROWS	11010
GST_PROP_SOLVER_ROOT_LPNZ	11011
GST_PROP_SOLVER_FINAL_PROWS	11012
GST_PROP_SOLVER_FINAL_PNZ	11013
GST_PROP_SOLVER_FINAL_LPROWS	11014
GST_PROP_SOLVER_FINAL_LPNZ	11015
GST_PROP_SOLVER_LOWER_BOUND	11016
GST_PROP_SOLVER_CPU_TIME	21000
GST_PROP_SOLVER_ROOT_TIME	21001
GST_PROP_SOLVER_ROOT_LENGTH	21002

D Error Codes

Error Code	Value
GST_ERR_UNDEFINED	1000
GST_ERR_LIBRARY_CLOSED	1001
GST_ERR_PROPERTY_NOT_FOUND	1002
GST_ERR_PROPERTY_TYPE_MISMATCH	1003
GST_ERR_BACKTRACK_OVERFLOW	1004
GST_ERR_SOLUTION_NOT_AVAILABLE	1005
GST_ERR_RANK_OUT_OF_RANGE	1006
GST_ERR_INVALID_METRIC	1007
GST_ERR_NO_EMBEDDING	1008
GST_ERR_ALREADY_CLOSED	1009
GST_ERR_LP_SOLVER_ACTIVE	1010
GST_ERR_LOAD_ERROR	1011
GST_ERR_INVALID_NUMBER_OF_TERMINALS	1012
GST_ERR_PARAMETER_VALUE_OUT_OF_RANGE	1013
GST_ERR_UNKNOWN_PARAMETER_ID	1014
GST_ERR_INVALID_PROPERTY_LIST	1015
GST_ERR_INVALID_HYPERGRAPH	1016
GST_ERR_INVALID_NUMBER_OF_VERTICES	1017
GST_ERR_INVALID_NUMBER_OF_EDGES	1018
GST_ERR_INVALID_EDGE	1019
GST_ERR_INVALID_VERTEX	1020
GST_ERR_INVALID_DIMENSION	1021
GST_ERR_NO_STEINERS_ALLOWED	1022
GST_ERR_INVALID_CHANNEL	1023
GST_ERR_INVALID_CHANNEL_OPTIONS	1024
GST_ERR_INVALID_PARAMETERS_OBJECT	1025
GST_ERR_INVALID_PARAMETER_TYPE	1026
GST_ERR_EFST_GENERATOR_DISABLED	1029
GST_ERR_RFST_GENERATOR_DISABLED	1030
GST_ERR_UFST_GENERATOR_DISABLED	1031
GST_ERR_FST_PRUNER_DISABLED	1032
GST_ERR_INVALID_SOLVER	1033

E FST Data File Formats

The FST generators produce output called FST data files. (They are sometimes called “phase 1 data files”, since FST generation is the first phase of the two-phase process for computing Steiner trees.

FST data files come in three different formats, distinguished by version numbers. Currently there are three such formats corresponding to versions 0, 2 and 3 of the FST data format. (Version 1 is very obsolete, and no longer supported.)

Note that version 0 and 3 data formats can be used to describe Steiner tree in graph (or hypergraph) instances. However, GeoSteiner 5.2 *cannot* solve such problems. It blindly assumes all vertices are terminals. If given such an instance, GeoSteiner will produce the MST (i.e., the minimum tree spanning *all* vertices, be they terminals, Steiner vertices, or any mixture thereof.)

Version 0

Version 0 is used to represent an abstract MST or Steiner tree in graph or hypergraph problem instance. It is essentially the same format as used in Beasley’s “OR-library” – but extended slightly to handle hypergraph instances as well as graph instances. The OR-library format is as follows:

```
<Number of vertices N> <Number of edges M>
For each edge:
    <Vertex 1> <Vertex 2> <Edge cost>
<Number of terminal vertices K>
    <Terminal 1> <Terminal 2> ... <Terminal K>
```

Vertices are numbered 1 through N. Each `<Terminal i>` is the vertex number of a vertex that is a terminal (i.e., must be connected). The `<Edge cost>`’s are real numbers.

We extend this format slightly by permitting each edge to have two *or more* vertices. In exchange for this flexibility, we require the entire description of each edge to reside on a single line of the data file. Therefore, the final number on each line represents the hyperedge cost, and all preceding numbers on the line represent the vertices of the hyperedge.

Version 2

Version 2 is used primarily to represent geometric FSTs (Euclidean or rectilinear), although it can also handle non-geometric (graph) instances. It is unable, however, to represent Steiner trees in hypergraph problems, because it assumes every vertex is a terminal.

In the following description, fields enclosed in `<<...>>` are omitted when the Metric is Graph. The format is as follows:

```
<Version Number: literally "V2">
<Instance description (free text)>
<Metric: 1 = Rectilinear, 2 = Euclidean, 3 = Graph>
<Number of terminals (N)>
<<Decimal length of MST>> <<Hex length of MST>>
<<Number of duplicate terminal groups (ndg)>>
<Coordinate/length scaling factor>
<Machine description (free text)>
<Front-end CPU-time (1/100s of a second (integer number)>
<Number of hyperedges/FSTs (M)>
For each terminal:
    <<Dec X-coord>> <<Dec Y-coord>> <<Hex X-coord>> <<Hex Y-coord>>
For each duplicate terminal group:
    <<Number of duplicate terminals>>
    <<Terminal indices (1..N), on one line separated by spaces>>
For each hyperedge/FST:
    <Number of terminals (Ni)>
    <Terminal indices (1..N), on one line separated by spaces>
    <Decimal length of hyperedge/FST> <Hex length of hyperedge/FST>
    <<Number of Steiner points (Mi)>>
    For each Steiner point:
       <<Dec X-coord>> <<Dec Y-coord>> <<Hex X-coord>> <<Hex Y-coord>>
    <<Number of FST edges (Ki)>>
    For each FST edge:
       <<endpoint-1>> <<endpoint-2>>
    <FST status: 0 = never needed, 1 = maybe needed, 2 = always needed>
    <Number of incompatible FSTs>
    <Incompatible FST indices (1..M), on one line separated by spaces>
    <Number of concatenation terminals>
    <Conc. terminals indices (1..N), on one line separated by spaces>
```

Version 3

Version 3 is the default format, and represents geometric FSTs (Euclidean or rectilinear) as well as graph instances. Since it separately specifies each vertex to be either a terminal or Steiner vertex, it can also represent Steiner problems in graphs/hypergraphs. A number of obsolete fields from version 2 is omitted, however.

In the following description, fields enclosed in `<<...>>` are omitted when the Metric is Graph. The format is as follows:

```
<Version Number: literally "V3">
<Instance description (free text)>
<Metric: 1 = Rectilinear, 2 = Euclidean, 3 = Graph>
<Number of terminals (N)>
<<Decimal length of MST>> <<Hex length of MST>>
<Coordinate/length scaling factor>
<Decimal Integrality delta> <Hex Integrality delta>
<Machine description (free text)>
<Front-end CPU-time (1/100s of a second (integer number)>
<Number of hyperedges/FSTs (M)>
For each terminal:
    <<Dec X-coord>> <<Dec Y-coord>> <<Hex X-coord>> <<Hex Y-coord>>
For each terminal:
    <Terminal/Steiner flag: 0=Steiner, 1=Terminal>
For each hyperedge/FST:
    <Number of terminals (Ni)>
    <Terminal indices (1..N), on one line separated by spaces>
    <Decimal length of hyperedge/FST> <Hex length of hyperedge/FST>
    <<Number of Steiner points (Mi)>>
    For each Steiner point:
       <<Dec X-coord>> <<Dec Y-coord>> <<Hex X-coord>> <<Hex Y-coord>>
    <<Number of FST edges (Ki)>>
    For each FST edge:
       <<endpoint-1>> <<endpoint-2>>
    <FST status: 0 = never needed, 1 = maybe needed, 2 = always needed>
    <Number of incompatible FSTs>
    <Incompatible FST indices (1..M), on one line separated by spaces>
```

The following conventions are observed in versions 2 and 3 of the FST data file format:

- Data input routines require only that the individual data fields are separated by one or more white-space characters (space, tab, newline, vertical tab, and form-feed are the white-space characters of ANSI C).
- Data output routines shall align items according to the schema above:
  - Schema fields that appear on separate lines shall be written on separate lines.
  - Schema fields that are all on one line shall be written all on one line.
  - The data shall be indented as shown by the schema.
  - Each indentation level shall be one "tab stop".
  - The implementor may freely choose the width of this "tab stop".
- The `<Instance description (free text)>` and `<Machine description (free text)>` fields shall each be a sequence of 0 to 79 characters. Each character in the sequence may be any printable ASCII character except newline.
- The `<on one line separated by spaces>` fields are permitted to span several lines, so long as the additional lines are each indented an additional "tab stop". The intent of this splitting is to fully pack lines without exceeding some column limit (e.g., 80 columns). If no data is to appear then the line is removed completely.
- All decimal fields shall be *unscaled* – just as in the original terminal coordinate input data.
- The hexadecimal fields shall be *scaled*. For example suppose that the `<Coordinate scaling factor>` is K. Then the following relationship is implied:

  ```
          <Dec X-coord> = <Hex X-coord> / 10**K
  ```

  where the equal sign is meant to imply "is within epsilon of". Scaling of data shall be at the discretion of the FST generator. For example the FST generator is permitted to always specify a scaling factor of zero – thereby disabling the scaling feature. Programs that read FST data files should not assume that the hex-values (scaled or otherwise) are all integral without first verifying the actual data values.

- The `<Decimal Integrality delta>` (`<Hex Integrality delta>`) fields represent an *unscaled* (*scaled*) lower bound on the amount by which two solutions of different lengths must differ. For Euclidean FSTs, this must always be 0. For rectilinear FSTs scaled to integer lengths this would be 1 (scaled value). For graphs with integer weights, this can also be 1. The branch-and-cut can use this to provide earlier cutoff of nodes that cannot reduce the upper bound.

- Let fields `<endpoint-1>` and `<endpoint-2>`, occur within an FST containing N terminals and M Steiner points. Let the field value be J. Then the interpretation of the endpoint field is as follows:

  $1 \leq J \leq N \Longrightarrow$ endpoint is the Jth terminal in the FST's list of terminals.

  $-M \leq J \leq -1 \Longrightarrow$ endpoint is the $-J$th Steiner point in the FST's list of Steiner points.

- (only applicable for version 2 of the FST data file format)
  Duplicate terminal groups (DTGs) identify subsets of the vertices having identical coordinates:

  – The size of each DTG shall be at least 2.
  – Each terminal may be listed in at most one DTG.
  – The terminal indices listed in a single DTG must be distinct.
  – The first terminal in each duplicate terminal group shall be referenced by at least one FST (having FST status $\neq 0$).
  – The remaining terminals in each duplicate terminal group shall NOT be referenced by any FST (having FST status $\neq 0$).

- If an FST has "never needed" status then the FST generator may output *any* incompatibility and concatenation terminal information, including no information at all (such information is redundant).

- The incompatible information shall NOT include the FST itself.

- The incompatible information need not include FSTs which are "never needed".

- The incompatible information need not include FSTs which share two or more terminals. It is assumed that programs that read FST data files are smart enough to know about such basic incompatibilities already. Omitting such incompatibilities can significantly reduce the size of the data file.

- The FST-graph for rectilinear FSTs must always be a “left-most” and “top-most” Hwang topology. If not, such FSTs will not appear to be Hwang topologies when plotted.
- A simple top-down traversal of each Euclidean FST-graph starting from the first terminal must yield the recursive equilateral-point structure of the FST. In this way, programs that read Euclidean FST data files are able to correctly compute the exact length of each FST in terms of algebraic numbers, if desired.

Index