Version 2.0

System Documentation

LOLA – Library of Location Algorithms[1]

Working Group Professor H. W. Hamacher
Fachbereich Mathematik
Universität Kaiserslautern

Coordinators: Horst W. Hamacher, Holger Hennes, Kathrin Klamroth, Martin C. Müller, Stefan Nickel and Anita Schöbel

Home-page: http:\\www.mathematik.uni-kl.de\~lola

E-mail: lola@mathematik.uni-kl.de

[1]Partially supported by a grant of the Deutsche Forschungsgemeinschaft.

Contents

Chapter 1

Introduction

Finding "good" locations for facilities becomes more and more important in modern industry. The selection of optimized sites is essential for the efficient realization of technical methods. Examples can be found in microelectronics, the location of machines in industrial plants, the location of warehouses or depots, the location of emergency facilities or of undesireable facilities such as incinerators, etc.

The software library LOLA is designed to solve location problems and to suggest an optimized site for the specified problem. LOLA provides a graphical user interface that allows its simple application in industrial projects as well as for demonstrations in high school and university teaching. Furthermore a programming interface allows the use of the program library of LOLA for the implementation of extended routines to solve individual location problems.

In this library, efficient algorithms are implemented to handle various types of location models. A part of the algorithms is known from the literature, whereas others are (and will be) results of current research. Most of the algorithms are based on theoretic results from graph theory, computational geometry and combinatorial optimization.

A *location problem* usually includes a set of *existing facilities* Ex_i, $i = 1, \ldots, M$, which may be situated for example in the plane or on a road network. The objective is to find one (or several) new facilities X, such that a given cost function, as for example the total travel expense, is minimized. For this purpose a weight w_i, $i = 1, \ldots, M$ can be associated with each existing facility Ex_i representing for example the demand of facility Ex_i. The most common objectives are to solve either the Median problem or the Center problem, i.e. to look for a new location that minimizes either the average travel cost or the maximum travel cost.

For a detailed survey about location theory see e.g. Love et al. 1988[1], Francis et al. 1992[2]

[1] R.F. Love, J.G. Morris and G.O. Wesolowsky. *Facilities Location: Models & Methods.* North Holland, New York, 1988.

[2] R.L. Francis, F. Leon, J. McGinnis and J.A. White. *Facility Layout and Location: An Analytical Approach, 2nd ed.* Prentice-Hall 1992.

or Hamacher 1995[3].

In LOLA a classification scheme for location problems is used. It has been developed by Hamacher and Nickel [4] and is described in Chapter 6. The graphical user interface (GUI) of LOLA, which is also based on this classification scheme, provides a detailed help manual to guide the user to the appropriate solution of his/her problem.

The development of LOLA is part of a larger project in location theory supported by the German Research Council (DFG) and therefore is still in process. Whereas in this version 2.0 planar location problems as described in Hamacher 1995[3] and network location problems as well as some discrete problems can be solved, further network and discrete location problems will be included in an upcoming version of LOLA.

We would like to thank the DFG for the financial support that made this project possible.

How to download LOLA

You can download a copy of the source–code and some executable files from the internet:

URL http://www.mathematik.uni-kl.de/~lola

LOLA–email

For bug–reports or suggestions for improvements and further developments feel free to contact

lola@mathematik.uni-kl.de

[3]H.W. Hamacher. *Mathematische Verfahren der Planaren Standortplanung.* Vieweg 1995.

[4]H.W. Hamacher and S. Nickel: "Classification of Location Models". Technical Report in Wirtschaftsmathematik No. 19, Universität Kaiserslautern, Department of Mathematics, 1996. To appear in *Location Science.*

Part I

Getting Started with LOLA

Chapter 2

A Guided Tour through the LOLA Frontend

In this chapter we give a detailed introduction to the frontend of LOLA enabling the solution of different location problems. Nevertheless, this is just a guided tour and not a complete description. For details, we refer to Chapter 7.

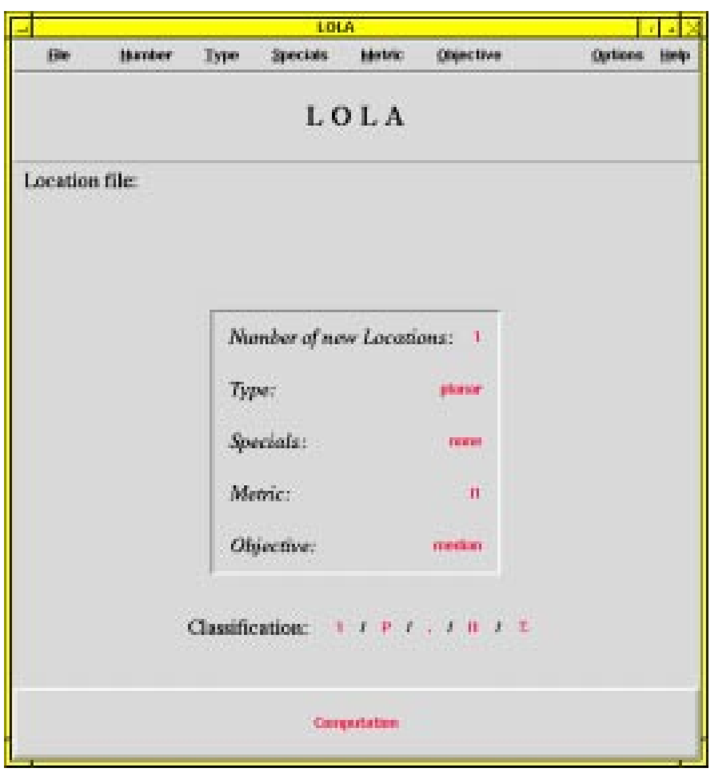

Figure 2.1: LOLA-frontend.

2.1 The classical 1-facility median problem

Assume that we want to solve the classical 1-facility-Weber problem, i.e. a planar minisum-problem with Euclidean distance. In the 5-position-classification scheme described in Chapter 6 this problem can be written as $1/P/./l_2/\sum$: We want to locate one new facility (1) in the plane (P) with no special assumptions (.), using the Euclidean distance as metric (l_2) and minimizing the sum of distances between the existing facilities and the new one ($\sum$).

In order to solve a problem of this type, follow steps a) to c).

a) Select the following options in the menue bar:

   - $\boxed{\text{Number}}$: select $\boxed{\text{1-facility}}$
   - $\boxed{\text{Type}}$: select $\boxed{\text{planar}}$
   - $\boxed{\text{Extras}}$: select $\boxed{\text{none}}$
   - $\boxed{\text{Metric}}$: select $\boxed{l2}$
   - $\boxed{\text{Objective}}$: select $\boxed{\text{median}}$

   The classification string on the LOLA screen should now look like

$$1/P/\cdot/l2/\sum.$$

b) Load a file with location data. The sample file "sample.loc " can be found in the directory

   .../LOLA/examples

   The file "sample.loc" contains the data of six existing facilities:

   begin {location} [2,1]

   15.00 16.00 955 [Koeln]

   10.00 36.00 577 [Duesseldorf]

   9.00 52.00 540 [Duisburg]

   20.00 53.00 620 [Essen]

   42.00 58.00 610 [Dortmund]

   48.00 90.00 261 [Muenster]

   end {location}

   The first line implies that we solve a problem in two dimensions with one objective function. The following lines contain, for every existing facility, the x-coordinate, the y-coordinate, the weight, and — optional — a symbolic name of the existing facility.

   To load the file click on $\boxed{\text{File}}$ in the menu bar and select $\boxed{\text{Load Location}}$. The resulting LOLA window is given in Figure 2.2.

   To generate your own data files you can use either

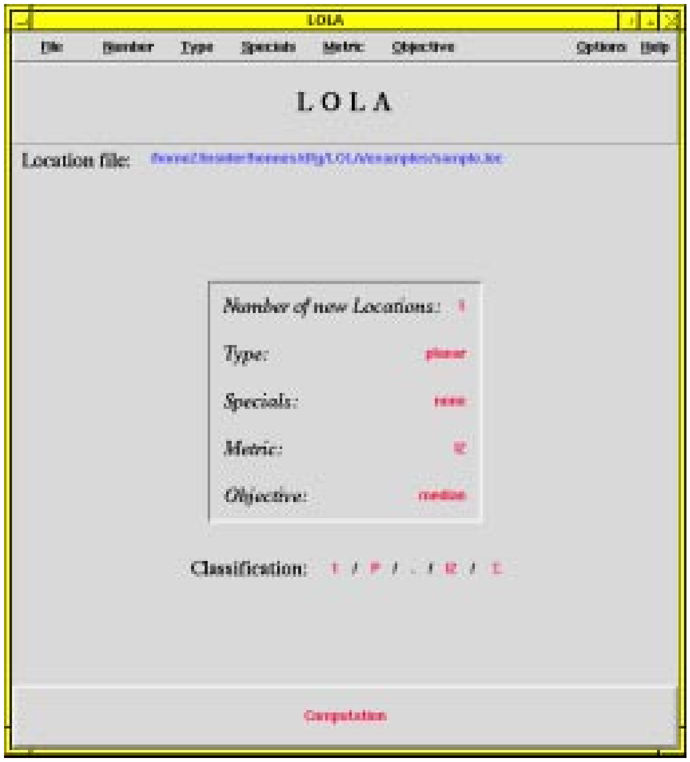

Figure 2.2: Loading a location file.

- any ASCII-editor to type in the data like in sample.loc according to the data file specifications, see Section 4.1.
- or the graphical editor that can be found under Graphical Edit in the File menu. In this editor the location data can be entered using the mouse.

c) Click on computation, and LOLA will provide a window with the solution *Opt*, see Figure 2.3.

   If you click on View Results you can see the coordinates and the objective value of the optimal solution.

2.2 A planar problem with polyhedral gauges

Now suppose that the Euclidean distance is not suitable to model the problem in our first example. Instead of the Euclidean distance function, we can use the option of LOLA to model distances by polyhedral gauges, a special case of which are block norms (polyhedral norms). In order to choose this option in the LOLA frontend, follow steps a) and b) below:

a) Choose the options in the menu analogous to the previous example, except for the metric: Under the option metric we select gauge which leads to the classification string $1/P/./gauge/\sum$ on the frontend.

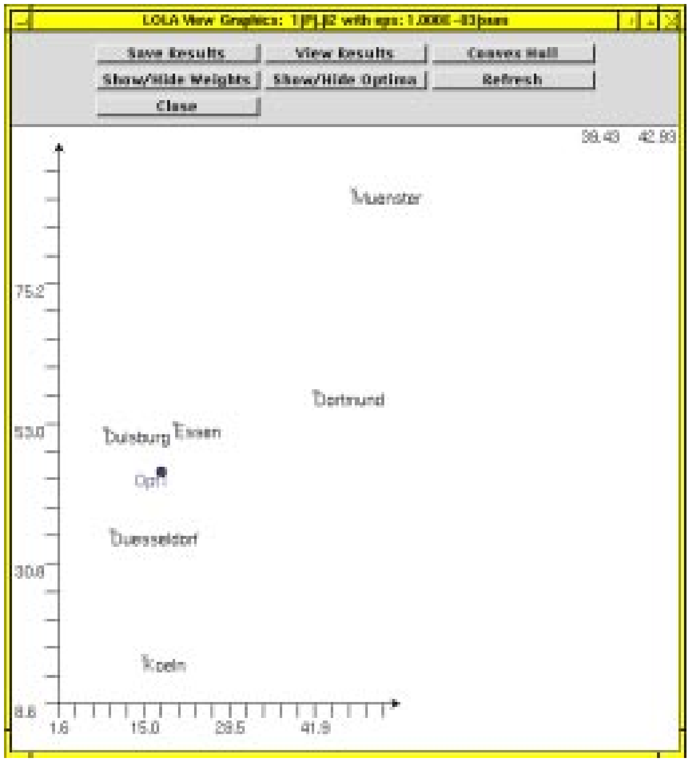

Figure 2.3: Solution-window for a problem of type $1/P/./l_2/\sum$.

b) In the next step we have to load a location file. In the case of gauge distances, the location file must provide more information since a different gauge may be used to measure the distance to each existing facility. Thus additionally to the facilities coordinates and weights, a number has to be specified corresponding to the respective gauges in a gauge-file. In Figure 2.4, an example of a gauge file "sample.gau" with two gauges is given, the first of which can be referred to as gauge(0) and the second as gauge(1). Their unit balls can be seen in Figures 2.5 and 2.6, respectively.

   Remark: A (polyhedral) gauge is defined by the coordinates of its extreme points in counterclockwise order. (Note that gauges always must be convex!)

   A corresponding location file "sample_gauge.loc" is given in Figure 2.7. Both files can be found in the directory

   .../LOLA/examples

   The solution window for this problem is depicted in Figure 2.8.

2.3 A network-problem

To solve network location problems we need the additional information of the network which is given in an adjacency matrix which is represented by an adjacency list in LOLA. This adjacency matrix defines the edge length of the network, i.e. the distances between pairs of

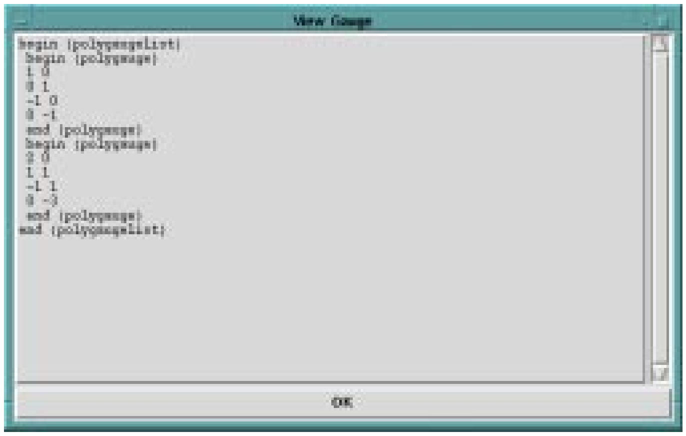

Figure 2.4: Example- gauge file with two gauges.

existing facilities, and is added to the location file. An example is given in Figure 2.9. (The corresponding location file "sample.gra" can be found in the directory .../LOLA/examples).

In this file, the existing facilities are specified and the adjacency matrix of the network G is represented in a list. Each row of this adjacency list contains the starting node, the end node and the length of the corresponding edge.

Additionally to the correct choice of the input file we have to select the option graph in the Type menu. Furthermore, in the menu Metric, we can choose between the options d(V,V) and d(V,G). Selecting d(V,V) restricts the search for an optimal solution to the nodes of the network (graph) G.

The solution window of the corresponding problem of type $1/G/./d(V,V)/\sum$ is given in Figure 2.10.

2.4 A discrete problem

For the case we want to solve a discrete problem we need more information about the type of the problem and the locations. At moment LOLA could solve the uncapacitated facility location problem (UFLP). Therefore we need information about the demand points, the supply points and the costs for moving from a supply point to demand point. In the example file "sample.dis" we see two list of facility and a cost matrix.

Before selecting the input file we have only to select option discrete in the Type menu. When pressing Computation we get a new window for choosing the type of the heuristic. We make three different heuristics and an exact algorithm for this problem available.

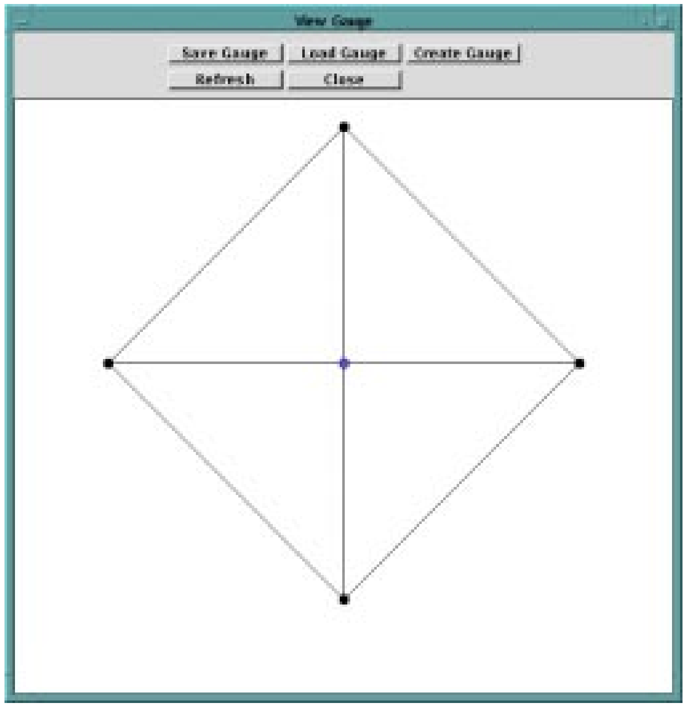

Figure 2.5: The unit ball of gauge(0) in the example file (cf. Figure 2.4).

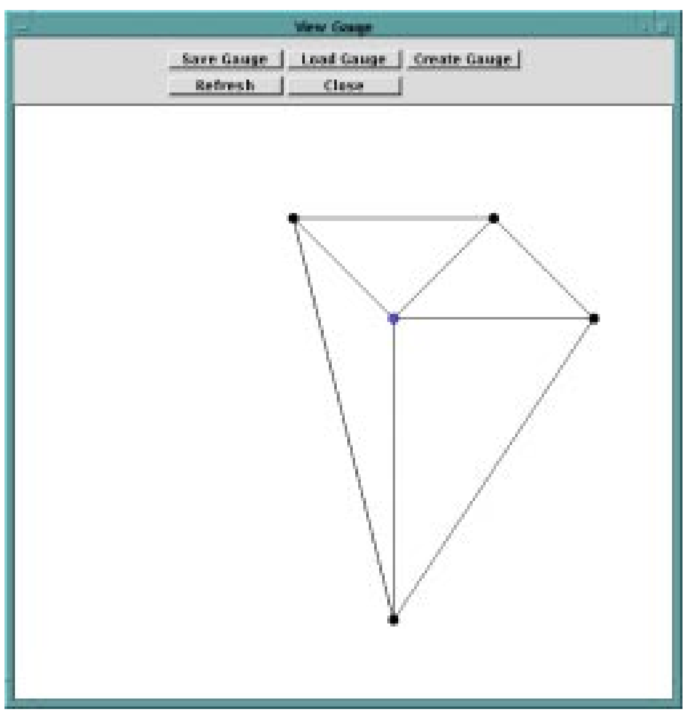

Figure 2.6: The unit ball of gauge(1) in the example file (cf. Figure 2.4).

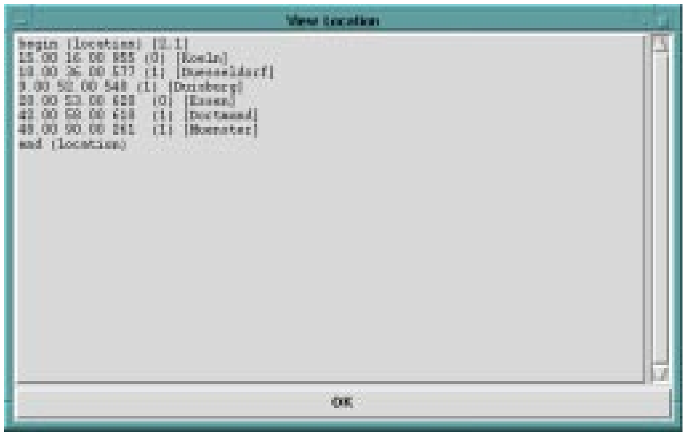

Figure 2.7: A location-file containing the specification (0) and (1) for the respective gauges.

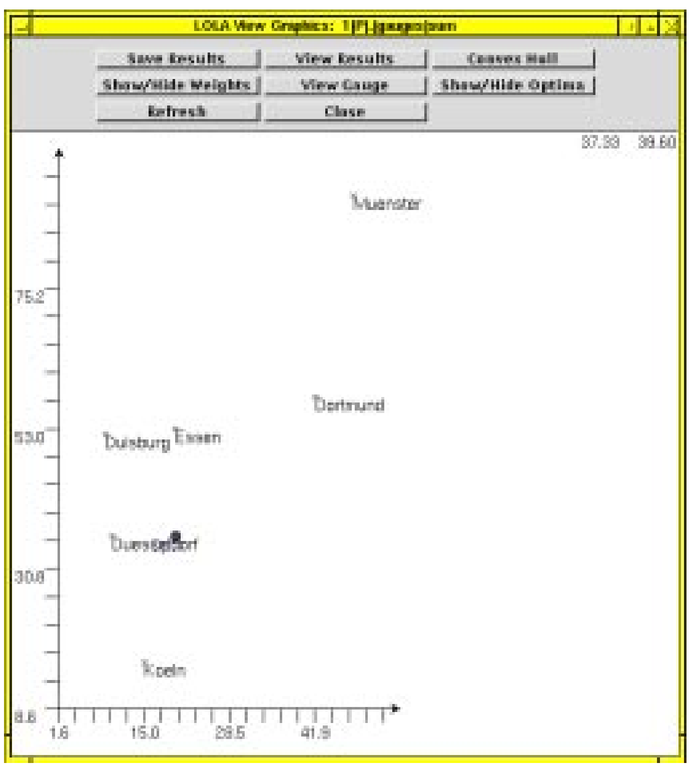

Figure 2.8: Solution window for the problem $1/P/./gauge/\sum$ defined above.

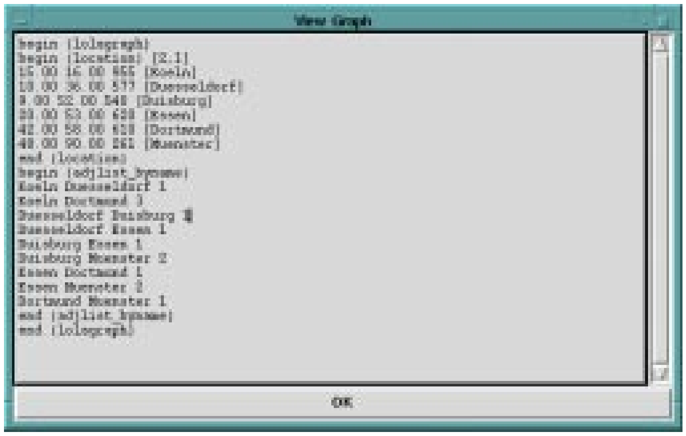

Figure 2.9: The input file for a network location problem.

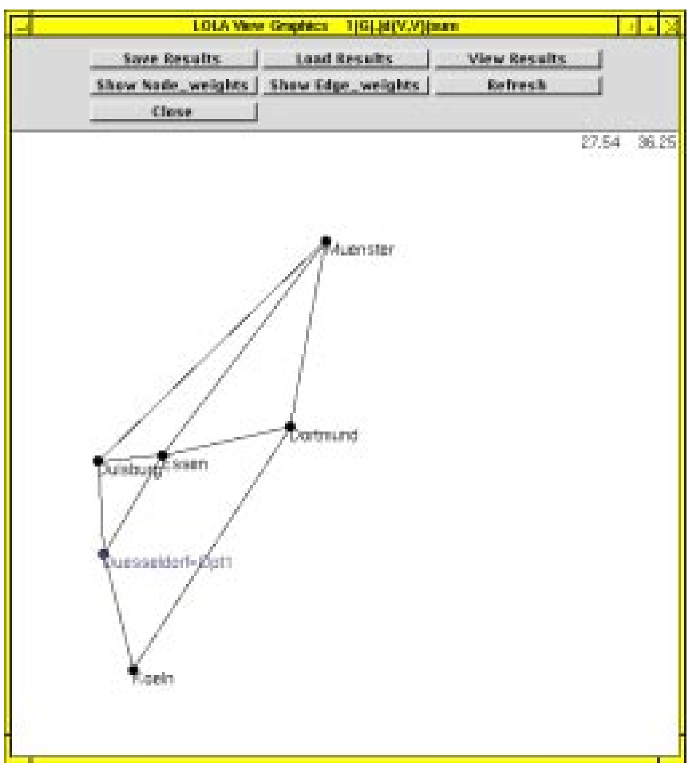

Figure 2.10: Solution window for the network problem $1/G/./d(V,V)/\sum$ defined above.

Chapter 3

Location Algorithms Available in LOLA

The algorithms available in LOLA are listed in Table 3.1.

Problem Class	Description of the Classification Scheme
$1/P/./l_1/\sum$	1-Median problem in the plane with l_1-distances.
$1/P/R/l_1/\sum$	Restricted 1-Median problem in the plane with l_1-distances and forbidden region inside.
$1/P/R^c/l_1/\sum$	Restricted 1-Median problem in the plane with l_1-distances and forbidden region outside.
$N/P/./l_1/\sum$	N-Median problem in the plane with l_1-distances.
$1/P/./l_1/2-\sum_{par}$	Bi-objective 1-Median problem in the plane with l_1-distances.
$1L/P/./l_1/\sum$	Locating 1 line in the plane wrt. l_1-distances and the median objective function.
$1L/P/R=convpolyhed/l_1/\sum$	Locating 1 line in the plane wrt. l_1-distances and the median objective function with convex polyhedral forbidden regions.
$1/P/./l_2/\sum$	1-Median problem in the plane with Euclidean distances.
$1/P/R=convpolyhed/l_2/\sum$	Restricted 1-Median problem in the plane with l_2-distances and forbidden region inside.
$1/P/B/l_2/\sum$	Restricted 1-Median problem in the plane with l_2-distances and barriers.
$N/P/./l_2/\sum$	N-Median problem in the plane with Euclidean distances.
$1L/P/./l_2/\sum$	Locating 1 line in the plane wrt. Euclidean distances and the median objective function.
$1L/P/w_i=1/l_2/\sum$	Locating 1 line in the plane wrt. Euclidean distances and the median objective function where all weights are equal to 1.
$1L/P/R=convpolyhed/l_2/\sum$	Locating 1 line in the plane wrt. Euclidean distances and the median objective function with convex polyhedral forbidden regions.
$1/P/./l_2^2/\sum$	1-Median problem in the plane with squared Euclidean distances.

$1/P/R/l_2^2/\sum$	Restricted 1-Median problem in the plane with squared Euclidean distances and forbidden region inside.
$1/P/R^c/l_2^2/\sum$	Restricted 1-Median problem in the plane with squared Euclidean distances and forbidden region outside.
$N/P/./l_2^2/\sum$	N-Median problem in the plane with squared Euclidean distances.
$1/P/./l_2^2/Q-\sum_{par}$	Q-objective 1-Median problem in the plane with squared Euclidean distances.
$1/P/./l_p/\sum$	1-Median problem in the plane with l_p-distances ($1 < p < \infty$).
$1/P/R = convpolyhed/l_p/\sum$	1-Median problem in the plane with l_p-distances with convex polyhedral forbidden regions.
$N/P/./l_p/\sum$	1-Median problem in the plane with l_p-distances.
$1L/P/./l_p/\sum$	Locating 1 line in the plane wrt. l_p-distances and the median objective function.
$1L/P/R = convpolyhed/l_p/\sum$	Locating 1 line in the plane wrt. l_p-distances and the median objective function with convex polyhedral forbidden regions.
$1/P/./l_\infty/\sum$	1-Median problem in the plane with l_∞-distances.
$1/P/R/l_\infty/\sum$	Restricted 1-Median problem in the plane with l_∞-distances and forbidden region inside.
$1/P/R^c/l_\infty/\sum$	Restricted 1-Median problem in the plane with l_∞-distances and forbidden region outside.
$N/P/./l_\infty/\sum$	N-Median problem in the plane with l_∞-distances.
$1/P/./l_\infty/2-\sum_{par}$	Bi-objective 1-Median problem in the plane with l_∞-distances.
$1L/P/./l_\infty/\sum$	Locating 1 line in the plane wrt. l_∞-distances and the median objective function.
$1L/P/R = convpolyhed/l_\infty/\sum$	Locating 1 line in the plane wrt. l_∞-distances and the median objective function with convex polyhedral forbidden regions.
$1/P/./\gamma/\sum$	1-Median problem in the plane with polyhedral gauges.
$1/P/./\gamma/2-\sum_{par}$	Bi-objective 1-Median problem in the plane with polyhedral gauges.
$1L/P/./\gamma_B/\sum$	Locating 1 line in the plane wrt. block norms and the median objective function.
$1L/P/R = convpolyhed/\gamma_B/\sum$	Locating 1 line in the plane wrt. block norms and the median objective function with convex polyhedral forbidden regions.
$1/P/./l_1/max$	1-center problem in the plane with l_1-distances.
$1/P/w_i = 1/l_1/max$	1-center problem in the plane with l_1-distances where all weights are equal to 1.
$1/P/R = convex/l_1/max$	1-center problem in the plane with l_1-distances with convex polyhedral forbidden regions.
$N/P/./l_1/max$	N-center problem in the plane with l_1-distances.
$1L/P/./l_1/max$	Locating 1 line in the plane wrt. l_1-distances and the

	center objective function.
$1/P/./l_2/max$	1-center problem in the plane with Euclidean distances.
$1/P/R/l_2/max$	Restricted 1-center problem in the plane with Euclidean distances.
$1L/P/./l_2/max$	Locating 1 line in the plane wrt. Euclidean distances and the center objective function.
$1L/P/./l_p/max$	Locating 1 line in the plane wrt. l_p-distances $(1 < p < \infty)$ and the center objective function.
$1/P/./l_\infty/max$	1-center problem in the plane with l_∞-distances.
$1/P/w_i = 1/l_\infty/max$	1-center problem in the plane with l_∞-distances where all weights are equal to 1.
$1/P/R = convex/l_\infty/max$	1-center problem in the plane with l_∞-distances with convex polyhedral forbidden regions.
$N/P/./l_\infty/max$	N-center problem in the plane with l_∞-distances.
$1L/P/./l_\infty/max$	Locating 1 line in the plane wrt. l_∞-distances and the center objective function.
$1/P/./\gamma/\max$	1-Center problem in the plane with polyhedral gauges.
$1/P/R/\gamma/\max$	1-Center problem in the plane with polyhedral gauges and forbidden regio
$1L/P/./\gamma_B/\max$	Locating 1 line in the plane wrt. block norms and the center objective function.
$1/G_D/./d(V,V)/\sum$	1-Median problem on a directed graph, restricted to the nodes of G_D.
$1/G/./d(V,V)/\sum$	1-Median problem on a graph, restricted to the nodes of G.
$1/G/./d(V,G)/\sum$	1-Median problem on a graph where the optimal solution may be anywhere on G.
$1/T/./d(V,V)/\sum$	1-Median problem on a tree T, restricted to the nodes of T.
$1/T/./d(V,T)/\sum$	1-Median problem on a tree T where the optimal solution may be anywhere on T.
$1/G/./d(V,G)/2-\sum_{par}$	Bi-objective 1-Median problem on a graph G.
$1/G/./d(V,G)/Q-\sum_{par}$	Q-objective 1-Median problem on a graph G.
$1/G/./d(V,V)/Q-\sum_{par}$	Q-objective 1-Median problem on a graph G, restricted to the nodes of G.
$1/G_D/./d(V,G)/Q-\sum_{par}$	Q-objective 1-Median problem on a directed graph G_D.
$1/G_D/./d(V,V)/Q-\sum_{par}$	Q-objective 1-Median problem on a directed graph G_D, restricted to the nodes of G_D.
$1/G/./d(V,G)/Q-\sum_{lex}$	Q-objective 1-Median problem on a graph G wrt. the lexicographic ordering where the optimal solution may be anywhere on the graph G.
$1/G/./d(V,V)/Q-\sum_{lex}$	Q-objective 1-Median problem on a graph G, restricted to the nodes of G wrt. the lexicographic ordering.
$1/G_D/./d(V,V)/Q-\sum_{lex}$	Q-objective 1-Median problem on a directed graph G wrt. the lexicographic ordering where the optimal solution may be anywhere on the graph.
$1/G_D/./d(V,V)/Q-\sum_{lex}$	Q-objective 1-Median problem on a directed graph G, restricted to the nodes of G wrt. the lexicographic ordering.

$1/G_D/./d(V,G)/\max$	1-Center problem on a directed graph G_D.
$1/G_D/./d(V,V)/\max$	1-Center problem on a directed graph, restricted to the nodes of G_D.
$1/G/./d(V,V)/\max$	1-Center problem on a graph G, restricted to the nodes of G.
$1/G/./d(V,G)/\max$	1-Center problem on a graph G where the optimal solution may be anywhere on G.
$1/T/./d(V,V)/\max$	1-Center problem on a tree T, restricted to the nodes of T.
$1/T/./d(V,T)/\max$	1-Center problem on a tree T where the optimal solution may be anywhere on T.
$1/G_D/./d(V,V)/Q-\max_{par}$	Q-objective 1-Center problem on a directed graph G_D, restricted to the nodes of G_D.
$1/G_D/./d(V,G)/Q-\max_{par}$	Q-objective 1-Center problem on a directed graph G_D where the optimal solution may be anywhere on the directed graph G_D.
$1/G/./d(V,V)/Q-\max_{lex}$	Q-objective 1-Center problem on a graph G, restricted to the nodes of G, wrt. the lexicographic ordering.
$1/G_D/./d(V,V)/Q-\max_{lex}$	Q-objective 1-Center problem on a directed graph G, restricted to the nodes of G, wrt. the lexicographic ordering.
$N/G/./d(V,V)/\sum$	N-Median problem on a graph G, restricted to the nodes of G.
$N/G/./d(V,V)/\max$	N-Center problem on a graph G, restricted to the nodes of G.
$\#/D/././.$	Uncapacitated facility location problem.

Table 3.1: Algorithms of LOLA

Chapter 4

Format of the Input Data

The system LOLA reads and interprets problem data for location problems in a descriptive language especially designed for that task. Furtheron a graphical editor is available to convert graphical input into that language.

Note that the blanks in the environment specifications as given below can not be ommitted!

4.1 File Type *loc* (containing location data)

begin {location} [d,Q]
$x_{11} \cdots x_{1d}$ $w_{11} \cdots w_{1Q}$ [symbolic name of facility 1]
$\vdots$ $\vdots$ $\vdots$
$x_{M1} \cdots x_{Md}$ $w_{M1} \cdots w_{MQ}$ [symbolic name of facility M]
end {location}

x_{ij} j-th coordinate of the i-th facility

w_{ij} For $\boxed{\text{N-Facilities}}$-problems: For each new facility there must be a weight representing the importance (demand) of this new facility wrt. the existing facilities. The value w_{ij} represents the weight of the j-th new facility wrt. the i-th existing facility. Note that in the case that $N = 1$, i.e. for 1-facility problems, only one weight w_i has to be specified for each existing facility.

For $\boxed{\text{Q-median}}$ or $\boxed{\text{Q-center}}$ problems: w_{ij} represents the weight (demand, importance) of the existing facility i with respect to the j-th criterion (objective).

d: Dimension of the facilities.

Q: For $\boxed{\text{N-Facilities}}$-Problems: Q is the dimension of the weights, which is in this case equal to the number N of new locations.

For $\boxed{\text{Q-median}}$ or $\boxed{\text{Q-center}}$ problems: The number of criteria (objectives) according to which the problem is to be solved.

As an example consider a planar $\boxed{\text{1-facility}}$-problem with Median-Objective. Then $[d, Q]$ is assigned the values [2,1], i.e. we consider a problem in two dimensions and we need one weight per existing facility in order to find the new location. When the graphical editor is used, the weight is set to 1 by default.

4.2 File Type *mat* (containing the information about interactions between the new facilities in case of $\boxed{\textbf{N-Facilities}}$ problems)

begin {matrix} [N]
$w_{11} \cdots w_{1N}$
$\vdots$
$w_{N1} \cdots w_{NN}$
end {matrix}

The entries w_{ij} represent the interaction (weight) between the new locations. The value N specifies the number of new facilities sought. Note that a matrix file of this type is only needed for $\boxed{\text{N-Facilities}}$ problems where $N \geq 2$.

4.3 File Type *res* (containing the information about restrictions)

In case of two dimensional, planar location problems, restrictions (forbidden regions or barriers for the new locations) can be introduced to the problem using files of type *res*:

begin {restriction} [2]
begin {polyhedron} [m]
x_1 y_1 (x_i, y_i) represent the (two-dimensional) coordinates
x_2 y_2 of the i-th extreme point of a polyhedron.
$\vdots$ Note that this polyhedron may also be non-convex.
x_n y_n
end {polyhedron}

begin {conpolyhedron} [m]
x_1 y_1 This environment should be alternatively used in case
x_2 y_2 of convex polyhedral restrictions.
$\vdots$
x_n y_n
end {conpolyhedron}

`begin {circle} [m]`
$x\ y\ r$ The point (x, y) specifies the center-point of the circle and r its radius.
`end {circle}`

`begin {rectangle} [0]`

$x_1\ y_1\ \Delta x\ \Delta y\ \alpha$	In the case that a restriction is represented by a rectangle, (x_1, y_1) specifies the lower left corner of the rectangle. Δx and Δy specify the height and width, respectively, and α is the angle between the rectangles lower side and the x-axis.

`end {rectangle}`

`begin {segment} [inf]`
$x_1\ y_1$ A segment is represented by its two end points
$x_2\ y_2$ (x_1, y_1) and (x_2, y_2).
$\vdots$
$x_n\ y_n$
`end {segment}`

`end {restriction}`

Remark The coefficient m indicates the type of restriction:

0: restriction is a forbidden region.

inf: restriction is a barrier.

Example In the following example the input data for a 1-facility problem in the plane ($\mathbb{R}^2$) is given where additionally a circular restriction is taken into account. The location file of type *loc* is given by

```
begin {location} [2,1]
2 2 1 [OrtA]
5 3 1 [OrtB]
8 3 1 [OrtC]
4 4 1 [OrtD]
end {location}
```

If we select the option restrictions in the Specials menu and then the restriction type circle, a possible choice of a restriction-file is

```
begin {restriction} [2]
begin {circle} [0]
```

```
4 4 2
end {circle}
end {restriction}
```

In Figure 4.1 the solution window of a 1-median problem in the plane with the l_∞ metric is given using this input data.

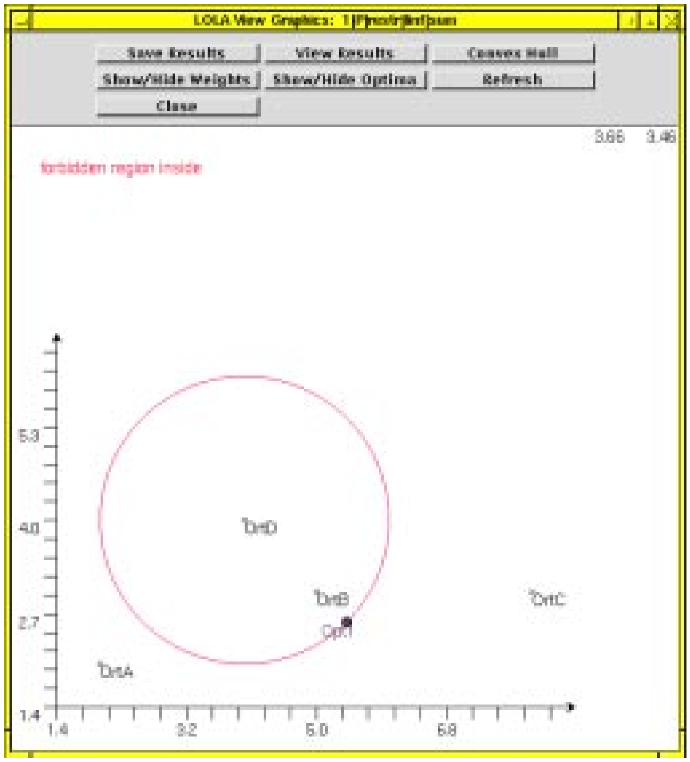

Figure 4.1: Solution window of a problem of type $1/P/R = C/l_\infty/\sum$ with a circular restriction.

Figure 4.2 shows the solution window of the same problem with a different (polyhedral) restriction.

Note that in the current version of LOLA only the above specified types of restrictions (i.e. polyhedral or circular sets) can be implemented. Nevertheless it is possible to include several restrictions of the same type into one restriction file whereas other types may be ommitted. The individual restriction regions have to be pairwise disjoint!

4.4 File Type *gauge* (containing the data of polyhedral gauges)

The input format for files of the type *polygauge* consists of a list of points which define a convex polyhedron containing the origin in its interior. The points in this list must be sorted in counterclockwise order. Several polyhedral gauges may be collected in a file of type *polygaugelist*.

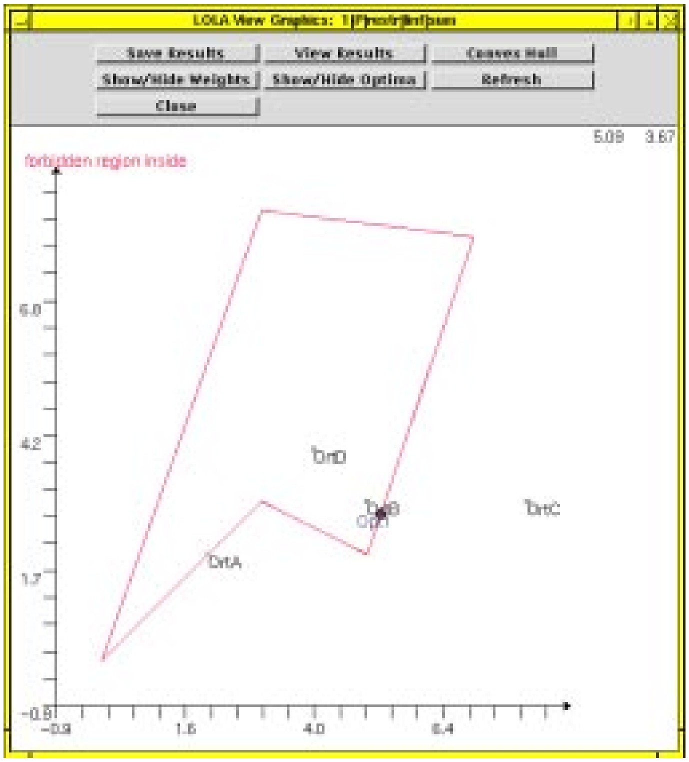

Figure 4.2: Solution window of a problem of type $1/P/R = P/l_\infty/\sum$ with a (non-convex) polyhedral restriction.

`begin {polygaugelist}`
`begin { polygauge}`
$x_1\ y_1$
$\vdots$
$x_n\ y_n$
`end {polygauge}`
$\vdots$
`begin {polygauge}`
$x_1\ y_1$
$\vdots$
$x_m\ y_m$
`end {polygauge}`
`end {polygaugelist}`

4.5 File Type *graph* (containing the location and network information for network problems)

The input format for network location problems consists of a *location*-file and a file representing the adjacency matrix of the network (graph) in an *adjacencylist*. For this purpose the format *adjlist* is provided where the edges of the corresponding network can be specified using their starting node (source node) and their end node (target node). The information of the existing facilities and the other nodes of the network is stored using the *location* format. Here the nodes can be specified by their (d-dimensional) coordinates which allows the use of location data from problems of planar type (cf. Section 4.1). They can be alternatively assigned the attribute NC, i.e. "no coordinates" have to be specified. Nodes of the network not representing an existing facility can be included in this list by setting their weights w_{ij} equal to zero.

1. **Coordinate format of the location file:**

   `begin {lolagraph}`
   `begin {location} [d,Q]`
   $x_{11} \cdots x_{1d}$ $w_{11} \cdots w_{1Q}$ [symbolic name of facility 1]
   $\vdots$ $\vdots$ $\vdots$
   $x_{M1} \cdots x_{Md}$ $w_{M1} \cdots w_{MQ}$ [symbolic name of facility M]
   `end {location}`

2. **No-coordinate format of the location file:**

   `begin {location} [d,Q]`
   NC $w_{11} \cdots w_{1Q}$ [symbolic name of facility 1]
   $\vdots$ $\vdots$ $\vdots$
   NC $w_{M1} \cdots w_{MQ}$ [symbolic name of facility M]
   `end {location}`

   The adjacency list, which is additionally needed for network location problems, contains the definition and the lengths of the edges. Two different options are available: The edges can be defined by the numbers of the source- and target node, or by the symbolic names of the respective nodes.

3. **Number-format of the adjacency list**

   `begin {adjlist}`
   $source_1\, target_1$ ew_1
   $\vdots$ $\vdots$
   $source_n\, target_n$ ew_n
   `end {adjlist}`

   ew_i: Length of the i-th edge running from

   $source_i$: number of the starting node of the i-th edge in the location file to

   $target_i$: number of the end node of the i-th edge in the location file.

4. **Symbolic name-format of the adjacency list**

   `begin {adjlist_byname}`
   $sourcename_1\, targetname_1$ ew_1
   $\vdots$ $\vdots$
   $sourcename_n\, targetname_n$ ew_n
   `end {adjlist_byname}`
   `end {lolagraph}`

ew_i: Length of the i-th edge running from

$sourcename_i$: symbolic name of the starting node of the i-th edge in the location file to

$target_i$: symbolic name of the end node of the i-th edge in the location file

Example If we choose the option $\boxed{\text{graph}}$ in the $\boxed{\text{Type}}$ menu and the option $\boxed{\text{d(V,V)}}$ in the $\boxed{\text{Metric}}$ menu, a data file for the corresponding network location problem of type $1/G/./d(V,V)/\sum$ is given in Figure 4.3. An alternative representation of the same data is shown in Figures 4.5 and 4.6. The solution of this problem is given in Figure 4.4.

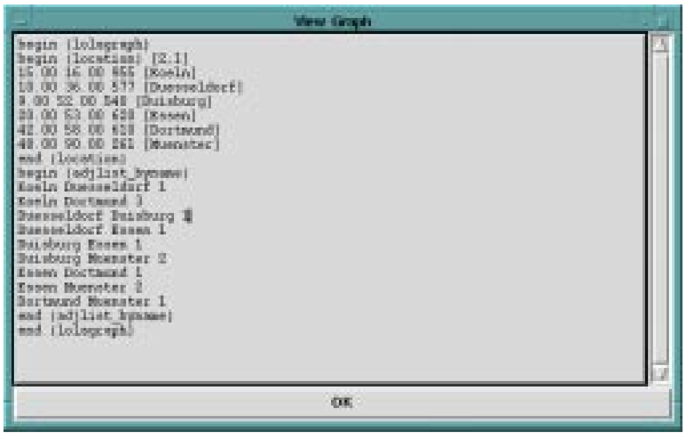

Figure 4.3: Input file for a network location problem - containing the location file in the coordinate-format and the adjacency matrix in the name-format.

The same solution we would get with the Input File shown in Figure 4.5.

4.6 File Type *dis* (containing data for discrete location problems)

An input format for a discrete location problem consists of three units: demand points, supply points and costs for moving from a supply point to a demand point. Each of the m demand points is specified by two coordinates and a weight for demand b. Optionally, a symbolic name may be given to a demand point.

begin {**discrete**}
begin {**demand**}
x_1 y_1 b_1 [symbolic name of demand point 1]
$\vdots$ $\vdots$ $\vdots$
x_m y_m b_m [symbolic name of demand point m]
end {**demand**}

Each of the n supply points is specified by two coordinates. However, the number of weights must be 1 or 2. This is given by a number p in the `begin` line. If there is only one weight

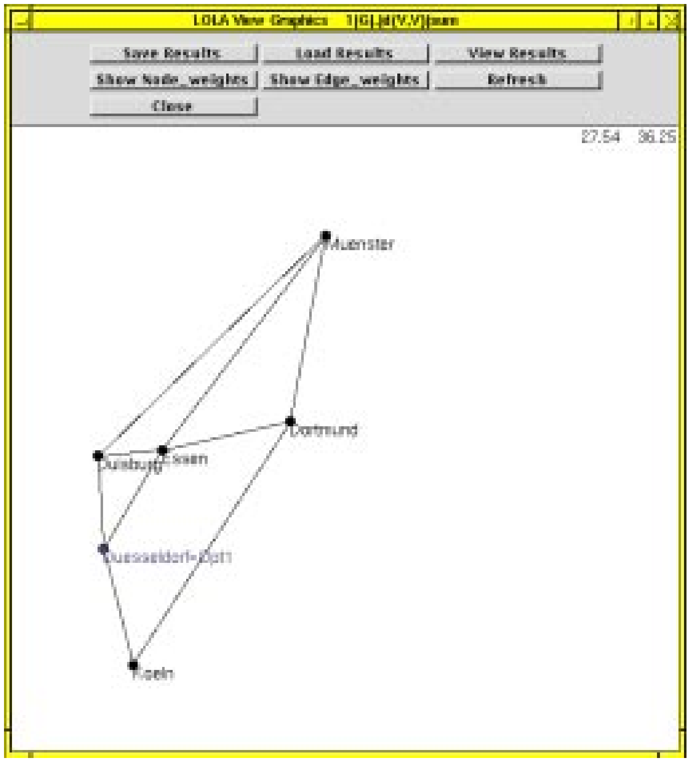

Figure 4.4: Solution window for the network location problem of type $1/G/./d(V,V)/\sum$ with the input data given in Figure 4.3.

then this is associated with the fixed costs for building a real supply depot in this location. The second weight represents the capacity constraints for the supply point.

begin {supply} [p]
x_1 y_1 f_1 a_1 [symbolic name of supply point 1]
$\vdots$ $\vdots$ $\vdots$
x_n y_n f_n a_n [symbolic name of supply point n]
end {supply}

If no list of supply points is given, then the demand points are at the same time supply points. Moreover, b_i denotes the fixed costs for location i.

The third input data is the $m \times n$ matrix for the costs.

begin {costmatrix}
c_{11} $\cdots$ c_{1n}
$\vdots$ $\vdots$ $\vdots$
c_{m1} $\cdots$ c_{mn}
end {costmatrix}
end {discrete}

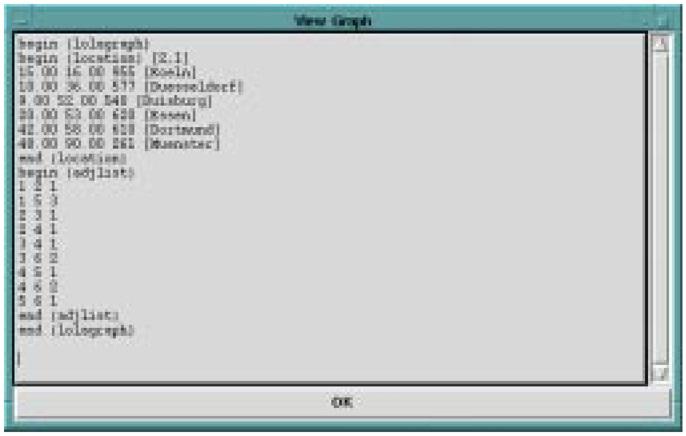

Figure 4.5: The same input file as in Figure 4.3 in the number-format.

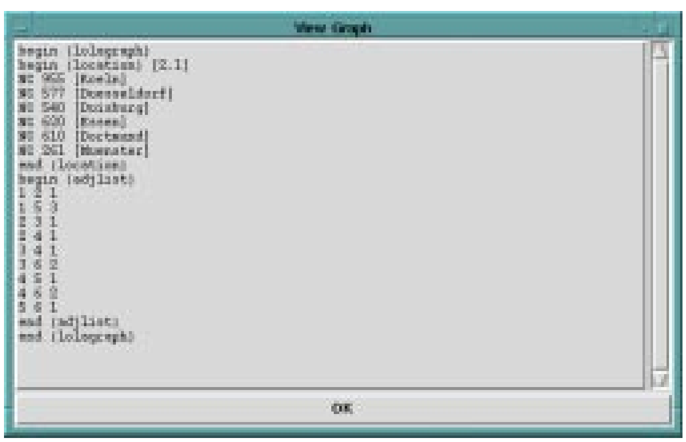

Figure 4.6: The input data of Figure 4.3 in the no-coordinate format.

4.7 File Type *sol* (containing solution data)

The information about the solution of a location problem is saved in files of type *sol*. Depending on the type of problem solved, this file may contain different information. In the first environment *class* the classification string of the solved problem is given. In the environment "objective value" the optimal objective value is specified whereas in the environment "polygonlist" one (or several, as e.g. in case of multicriteria problems) sets of optimal points/polyhedrons can be given.

begin {class}
$classification$
end {class} begin {result}
begin {objective value}
$z\ z\ z$
end {objective value}
begin {polygonlist}
begin {polygon}
$x_1\ y_1$
$x_2\ y_2$
$\vdots$
$x_n\ y_n$
end {polygon}
$\vdots$
begin {polygon}
$x_1\ y_1$
$x_2\ y_2$
$\vdots$
$x_n\ y_n$
end {polygon}
end {polygonlist}
end {result}

Example In the following the input data and the solution file are given for a planar location problem of type $1/P/./l_\infty/\sum$.

```
begin {location} [2,1]
2 2  1  [OrtA]
5 3  1  [OrtB]
8 3  1  [OrtC]
4 4  1  [OrtD]
end {location}
```

```
begin {class}
 $1/P/./l_{\infty}/\sum$
end {class}
begin {result}
```

```
begin {objective value}
7
end {objective value}
begin {polygonlist}
begin {polygon}
5 3
4 4
end {polygon}
end {polygonlist}
end {result}
```

Chapter 5

Writing a LOLA Application

In the following we will explain briefly how the LOLA-libraries can be used directly in a C++ program without using the LOLA frontend. We describe the components of a C++ program using LOLA to solve a planar minisum problem with squared Euclidean distance and restrictions.
First we have to include the definitions of the routines we need. In our case we need the routines that handle location files (read, write) and the algorithms for planar problems.

```
#include <LOLA/facs_util.h>
#include <LOLA/planealg.h>
```

Next we have to define some variables to store the objective value, the name of the location file (e.g. .../LOLA/examples/prog/test.loc), etc.

```
main() {
  double objval[2];
  string normact;
  string locfile="test.loc";
  string extrafile="test.restr";
  facs_util EX;
  planealg A;
  restrictions Restr;
```

Now we read the data for the problem.

```
  ifstream file (locfile,ios::in);
  EX.ReadLoc(file);
```

All necessary data is available to solve the corresponding unrestriced problem, which is done in the next step. Also the optimal solution is printed and the solution is shown graphically.

```
objval[0] = A.l2sqr_sum(EX);
A.WriteOpt();
list<location> AlgSol = A.alg_solution();
normact = "l2sqr";
EX.View(objval[0],normact,AlgSol,Restr);
```

Next we read in addition a restriction file and solve the corresponding restricted problem.

```
ifstream filerestr (extrafile,ios::in);
Restr = ReadRestr(filerestr);

objval[1] = A.l2sqr_sum(EX,Restr);
A.WriteOpt();
```

Finally, we show again the data, the restriction and the solution graphically in a window.

```
AlgSol = A.alg_solution();
EX.View(objval[1],normact,AlgSol,Restr);
```

The resulting complete C++ file given in the following can be compiled after LOLA is installed.

```
#include <LOLA/facs_util.h>
#include <LOLA/planealg.h>

main() {
  double objval[2];
  string normact;
  string locfile="test.loc";
  string extrafile="test.restr";
  facs_util EX;
  planealg A;
  restrictions Restr;

/*******************************************
* read  location-file                      *
```

```
*******************************************/

  ifstream file (locfile,ios::in);
  EX.ReadLoc(file);

/*******************************************
* solve  problem 1/P/./l2**2/sum           *
*******************************************/

  objval[0] = A.l2sqr_sum(EX);
  A.WriteOpt();
  list<location> AlgSol = A.alg_solution();
  normact = "l2sqr";
  EX.View(objval[0],normact,AlgSol,Restr);

/*******************************************
* read restriction-file                    *
*******************************************/

  ifstream filerestr (extrafile,ios::in);
  Restr = ReadRestr(filerestr);

/*******************************************
* solve problem 1/P/R/l2**2/sum            *
*******************************************/

  objval[1] = A.l2sqr_sum(EX,Restr);
  A.WriteOpt();


/*******************************************
* Output in a LEDA-Window                  *
*******************************************/

  AlgSol = A.alg_solution();
  EX.View(objval[1],normact,AlgSol,Restr);

}
```

The file "test.cpp" containing this C++ code can be found in the directory .../*LOLA*/*examples*/*prog*, where also a makefile is provided.

At the end of this section we show how some of the C++ methods used above are defined.

```
double planealg::l2sqr_sum(facilities& EX)
{
  double result;
```

```
  result = EX.l2sqr_sum();
  solution = EX.alg_solution();
  return result;
}

double planealg::l2sqr_sum(facilities& EX, restrictions& R)
{
  double result;
  result = R.l2sqr_sum(EX);
  solution = R.alg_solution();
  return result;
}
```

Chapter 6

A Classification Scheme for Location Problems

In this chapter we describe the classification scheme for location problems used in the frontend of LOLA. A detailed description of this scheme can be found in Hamacher and Nickel [1].

The classification scheme has five positions:

$$Pos1/Pos2/Pos3/Pos4/Pos5\ .$$

The meaning of each position is described in the following.

Pos1 This position contains information about the number and the type of the new facilities.

Pos2 The type of the location problem with respect to the decision space. This entry should e.g. differentiate between continuous, network and discrete problems.

Pos3 In this position is room for describing particularities of the specific location problem. For example, information about the feasible solutions or about capacity restrictions can be included in this position.

Pos4 This position is devoted to the relation of new and existing facilities. This relation may be expressed by some distance function or simply by assigned costs.

Pos5 The last position contains a description of the objective function.

If we do not make any special assumptions in a position this is indicated by /./. For example, /./ in Position 5 means that we consider any objective function and /./ in Position 3 means that the standard assumptions for the problem described in the remaining four positions hold. For example in planar location problems /./ in Position 3 implies e.g. that

[1] H.W. Hamacher and S. Nickel: "Classification of Location Models". Technical Report in Wirtschaftsmathematik No. 19, Universität Kaiserslautern, Department of Mathematics, 1996. To appear in *Location Science*.

we have (as usual) positive weights for the existing facilities. In general we also assume by default that the objective function is to be minimized.

The following table gives an overview about the usage of the classification scheme.

Position	Meaning	Usage (Examples)
1	number of new facilities	
2	type of problem	**P** planar location problem **D** discrete location problem **G** location problem on a network
3	specials	$w_m = 1$ all weights are equal $\mathcal{R}$ or R a forbidden region
4	type of distance function	l_1 Manhattan metric γ a gauge $d(V,V)$ node to node distance $d(V,G)$ node to points of graph distance
5	type of objective function	$\sum$ Median problem max Center problem $Q-\sum$ Multicriteria (Pareto) median problem

Note that, due to font limitations, in the LOLA frontend some symbols may not look exactly like they do in this manual.

A list of possible symbols used in each position of the classification scheme is given in the following table.

Position 1	Position 2	Position 3	Position 4	Position 5
$n \in \{1,\ldots,N\}$	$\mathbb{R}^d$	$\mathcal{R}$	l_p	$\sum$
l	P	$\mathcal{F}$	γ	max
p	H	$\mathcal{B}$	γ_{pol}	CD
A	$\mathcal{G}$	$w_m = 1$	γ_{mix}	$\int_d$
C	$\mathcal{G}_D$	$w_m \neq 0$	$\Vert \cdot \Vert$	$\int_{d_1} \int_{d_2}$
R	$\mathcal{T}$	w_m : distribution	d_{Haus}	$Q-\sum_{par}$
T	D	w_m : RV	d_{inhom}	$Q-\sum_{lex}$
G		w_m : $f(\cdot)$	$d(\mathcal{V},\mathcal{V})$	$Q-\sum_{MO}$
$\sharp$		mc	$d(\mathcal{V},\mathcal{G})$	$Q-(\sum,\max)_{par}$
$\sharp,\sharp$		alloc	$d(\mathcal{V},\mathcal{T})$	$\sum_{comp}$
		cap	$d(\mathcal{G},\mathcal{V})$	$\sum_{uncov}$
		bdg	$d(\mathcal{T},\mathcal{V})$	$\sum_{cov}+\sum_{uncov}$
		d_{max}	$d(\mathcal{G},\mathcal{G})$	$\sum_{cov}$
		price	$d(\mathcal{T},\mathcal{T})$	QAP
		queue		$\sum_{ord}$
				$\sum_{prob}$
				$\max_{prob}$
				$\sum_{hub}$
				φ : property

Chapter 7

The Components of LOLA

7.1 GUI – Graphical User Interface

The GUI is based on the 5–position classification scheme for facility location problems introduced in Chapter 6 to easily specify the type of problem which is going to be solved by LOLA.

If TCL/TK is available on your system, calling LOLA creates the window depicted in Figure 2.1.

According to the classification scheme, the menu of the GUI contains buttons for

Number to select the number of new facilities - in case of N-facility problems with the Insert-number-window as shown in Figure 7.1.

Figure 7.1: Insert-number-window

Type to select the basic type (P,G,D,T) of the problem.

- P: planar problems
- G: graph problems
- D: discrete problems
- T: tree problems

Specials to select special assumptions for the set of solutions, which may be

- equal weights In case of equal weights, faster procedures are available for some types of location problems.
- restrictions With the option outside we can choose whether the forbidden zone is inside or outside the given restrictions. The restrictions can be:
  - polyhedron The restriction file must describe a (possibly non-convex) polyhedron.
  - conpolyhedron The restriction file must describe a convex polyhedron.
  - circle The restriction file must describe a circle.
  - rectangle The restriction file must describe a rectangle.
  - all The restriction file must describe one or more of the above four possibilities in arbitrary order and number.
- barriers Not implemented yet!
- none Default possibility : no restrictions in the problem

Metric to select the distance function. The options are

- l1 The l_1-norm is chosen.
- l2 The l_2-norm is chosen.
- l2**2 The l_2^2-norm is chosen.
- linf The l_∞-norm is chosen.
- lp The l_p-norm is chosen, where p can be selected in Options under Preferences.
- gauge The distance measure is a self-created gauge.
- block The distance measure is a block norm.

For Graph and Tree-algorithms we have the options

- d(V,V) The optimal points are searched only on nodes.
- d(V,G) The optimal points are searched on the entire graph.
- d(V,T) The optimal points are searched on the entire graph which is a tree.

Objective to select the type of objective function. The options are

- median if the sum of (weighted) distances should be minimized.
- center if the maximum (weighted) distance should be minimized.
- Q-median if the sum over all distances should be minimized with respect to more than one criterion (i.e. the dimension of the weights is bigger than 1).
- Q-center if the maximum (weighted) distance should be minimized with respect to more than one criterion (i.e. the dimension of the weights is bigger than 1).

Furthermore the GUI contains

File
- to select files providing data for given problems, e.g. Load Location,
- to view this data, e.g. View Location (**Remark:** This option is not to edit a file) or
- to create new data files, e.g. Graphical Edit and Create Example,

Help to call the online help–browser,

Options to set general preferences on the maximum number of iterations, default metric for l_p or other problem dependent settings.

The solution process of the classified problems can be started by clicking on the button Computation.

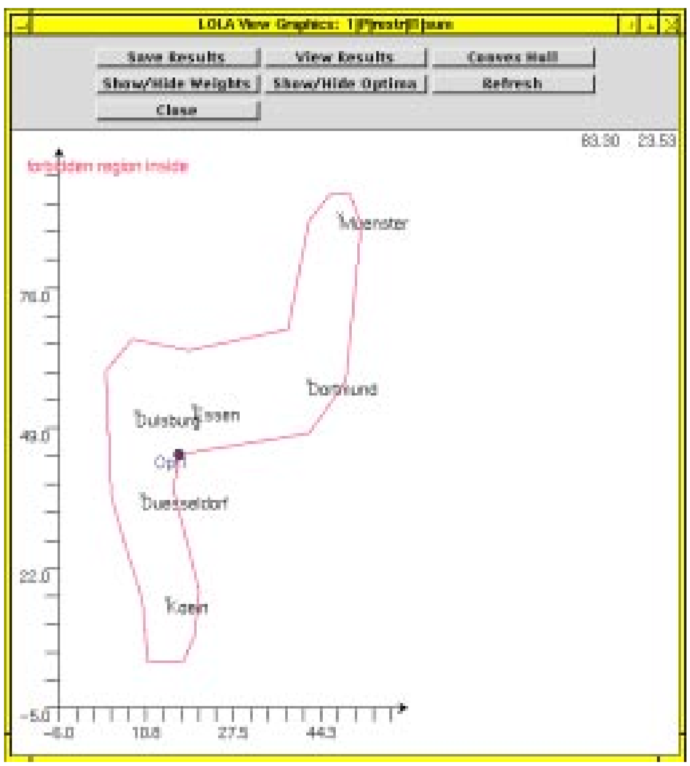

Figure 7.2: Solution window with a non-convex polyhedral restriction

A solution window (e.g. as shown Figure 7.2) could contain the following options:

Save Results to save the current solution,

View Results to view the results in numeric format, Refresh to redraw the solution window,

Close to close the window.

Solution windows for planar problems additionally contain the buttons

Convex Hull to draw, respectively remove the convex hull of the set of existing facilities,

Weights to show, respectively hide the weights of the existing facilities,

View Gauge to view the unit ball of a special gauge (appears only if the metric is set to gauge or block).

In solution windows for network problems the following buttons are available.

Node_Weights to show, respectively hide the weights of the edges of the network,

Edge_Weights to show, respectively hide the weights of the nodes of the network.

For discrete problems the following buttons are additionally available in the solution window.

Fix Costs to show, respectively hide the fix costs for the supply points,

Weights to show, respectively hide the weights of the demand points,

View Cost Matrix to view the cost matrix (advisable only for small problems).

7.2 Text Based Version of LOLA

All algorithms of LOLA can also be adressed using command-line options in the *text based mode*. TCL/TK is not needed for the text-based mode. The necessary input to solve a location problem can be given. Upon invocation LOLA returns the solution in text or graphical format, however the latter can be completely suppressed, rendering LOLA capable of operating text-only. This mode of operation is well suited to perform automated or repeated tasks.

The text-based mode is automatically entered if command-line options are detected and it is the only available mode if LOLA has been configured with `--withtcltk=no`. Figure 7.3 shows the available options.

The single-hyphen options detail which task LOLA is to perform, whereas `--output` specifies how to present the solution. The option `--output` allows a comma-seperated list of arguments, which are parsed from left to right:

```
lola -a <algorithm> [-l|-r|-g|-d|-m <file>] [-p <n>] [-n <n>] [--output=<arguments>]

Options:

  -a :  the algorithm, which is to be performed on the data (see below)
  -l : <file> contains the existing facilities
  -r : <file> contains restrictions
  -g : <file> contains a (directed or undirected) graph
  -d : <file> contains polygonal gauge definitions
  -m : <file> contains a cost matrix for n-facility problems
  -p : <n> is the exponent for an l_p-norm
  -n : <n> is the number of new facilities for problems on graphs
  --output:
      this option takes a comma-seperated list of arguments:
      [no]windowed : [dont] present the solution graphically
      [file=]<file>: write solution as text into <file>
      only the rightmost argument of each type takes effect
```

Figure 7.3: Command line options of the text-based version of LOLA

no `--output`	The solution is presented graphically in a window and text-based on standard output.
`--output=prob1.sol`	The solution is saved in the file `prob1.sol`
`--output=nowindowed`	No windows pop up – this enforces a *text-only* operation
`--output=file1,file=windowed`	The solution is saved into the file `windowed`, since the second argument supercedes the first. A solution window is generated.

-a <algorithm>	-l	-m	-r	-p	-g	-d	-n
l1_max, l2_max, linf_max,	X						
l1_v1_max, linf_v1_max, l2sqr_qsum	X						
l1_sum, l2_sum, l2sqr_sum, linf_sum	X						
l1_2sum, linf_2sum	X						
lp_sum	X		O	X			
in_l1_sum, out_l1_sum, in_l2sqr_sum,	X		X				
out_l2sqr_sum, in_linf_sum,	X		X				
out_linf_sum, in_convPoly_v1_l2_max,	X		X				
in_convex_linf_max, in_convex_l1_max	X		X				
barr_l2_sum	X		X				
N_l1_sum_v1, N_l2sqr_sum, N_l2_sum_v1,	X	X					
N_linf_sum_v1, N_l1_max, N_linf_max	X	X					
dir_median, undir_median,					X		
abs_undir_median					X		
abs_tree_median, abs_undir_center,					X		
abs_dir_center,loc_tree_center,					X		
abs_tree_center,loc_tree_median,					X		
loc_dir_center,loc_undir_center					X		
N_median_cplex, N_median_partitioning,					X		X
N_median_exchange, N_median_greedy,					X		X
N_center_partitioning, N_center_greedy					X		X
medPareto, loc_medPareto, loc_cenPareto,					X		
dir_medPareto, dir_locmedPareto,					X		
dir_loc_cenPareto, dir_cenPareto					X		
medLexi, loc_medLexi, loc_cenLexi,					X		
dir_medLexi, dir_loc_medLexi,					X		
dir_loc_cenLexi					X		
gauge_median, bi_crit_gauge_median	X					X	
gauge_center	X					X	
L_l1_sum_M3, L_linf_sum_M3,	X						
L_l2_sum_M3, L_l2_sum_M2logM,	X						
L_l2_sum_M2,	X						
L_lp_sum_M3, L_lp_max_M4	X			X			
L_l1_max_M4, L_linf_max_M4,	X						
L_l2_max_M4, L_l2_max_MlogM,	X						
L_l2_max_M2logM,	X						
L_block_sum_M3, L_block_max_M3	X					X	
L_RkonvPoly_l2_sum, L_RkonvPoly_l1_sum,	X		X				
L_RkonvPoly_lp_sum,	X		X	X			
L_RkonvPoly_linf_sum,	X		X				
L_RkonvPoly_block_sum	X		X			X	

Table 7.1: Feasible combinations of command-line options

argument of -a	problem class	LOLA method
l1_sum	$1/P/./l_1/\sum$	planealg::l1_sum
in_l1_sum	$1/P/R/l_1/\sum$	planealg::l1_sum
out_l1_sum	$1/P/R^c/l_1/\sum$	planealg::l1_sum
N_l1_sum_v1	$N/P/./l_1/\sum$	planealg::N_l1_sum
l1_2sum	$1/P/./l_1/2-\sum$	planealg::l1_2sum
L_l1_sum_M3	$1L/P/./l_1/\sum$	planealg::L_l1_sum_M3
L_RkonvPoly_l1_sum	$1L/P/R = convpolyhed/l_1/\sum$	planealg::L_RkonvPoly_l1_sum
l2_sum	$1/P/./l_2/\sum$	planealg::l2_sum
barr_l2_sum	$1/P/B/l_2/\sum$	planealg::l2_sum
N_l2_sum_v1	$N/P/./l_2/\sum$	planealg::N_l2_sum_v1
L_l2_sum_M2logM	$1L/P/./l_2/\sum$	planealg::L_l2_sum_M2logM
L_l2_sum_M3		planealg::L_l2_sum_M3
L_l2_sum_M2	$1L/P/w_i = 1/l_2/\sum$	planealg::L_l2_sum_M2
L_RkonvPoly_l2_sum	$1L/P/R = convpolyhed/l_2/\sum$	planealg::L_RkonvPoly_l2_sum
l2sqr_sum	$1/P/./l_2^2/\sum$	planealg::l2sqr_sum
in_l2sqr_sum	$1/P/R/l_2^2/\sum$	planealg::l2sqr_sum
out_l2sqr_sum	$1/P/R^c/l_2^2/\sum$	planealg::l2sqr_sum
N_l2sqr_sum	$N/P/./l_2^2/\sum$	planealg::N_l2sqr_sum
l2sqr_qsum	$1/P/./l_2^2/Q-\sum$	planealg::l2sqr_qsum
lp_sum	$1/P/./l_p/\sum$	planealg::lp_sum
lp_sum	$1/P/R = convpolyhed/l_p/\sum$	planealg::lp_sum
N_lp_sum_v1	$N/P/./l_p/\sum$	planealg::N_lp_sum_v1
L_lp_sum_M3	$1L/P/./l_p/\sum$	planealg::L_lp_sum_M3
L_RkonvPoly_lp_sum	$1L/P/R = convpolyhed/l_p/\sum$	planealg::L_RkonvPoly_lp_sum
linf_sum	$1/P/./l_\infty/\sum$	planealg::linf_sum
in_linf_sum	$1/P/R/l_\infty/\sum$	planealg::linf_sum
out_linf_sum	$1/P/R^c/l_\infty/\sum$	planealg::linf_sum
N_linf_sum_v1	$N/P/./l_\infty/\sum$	planealg::N_linf_sum
linf_2sum	$1/P/./l_\infty/2-\sum$	planealg::linf_2sum
L_linf_sum_M3	$1L/P/./l_\infty/\sum$	planealg::L_linf_sum_M3
L_RkonvPoly_linf_sum	$1L/P/R = convpolyhed/l_\infty/\sum$	planealg::L_RkonvPoly_linf_sum
gauge_median	$1/P/./\gamma/\sum$	gaugealg::sum
bi_crit_gauge_median	$1/P/./\gamma/2-\sum_{par}$	gaugealg::bi_crit_sum
L_block_sum_M3	$1L/P/./\gamma_B/\sum$	gaugealg::L_block_sum_M3
L_RkonvPoly_block_sum	$1L/P/R = convpolyhed/\gamma_B/\sum$	gaugealg::L_RkonvPoly_block_sum
l1_max	$1/P/./l_1/max$	planealg::l1_max
l1_v1_max	$1/P/w_i = 1/l_1/max$	planealg::l1_v1_max
in_convex_l1_max	$1/P/R = convex/l_1/max$	planealg::l1_max
N_l1_max	$N/P/./l_1/max$	planealg::N_l1_max
L_l1_max_M4	$1L/P/./l_1/max$	planealg::L_l1_max_M4
l2_max	$1/P/./l_2/max$	planealg::elzhearn
in_l2_max	$1/P/R/l_2/max$	planealg::l2_max
L_l2_max_M4	$1L/P/./l_2/max$	planealg::L_l2_max_M4
L_l2_max_MlogM	$1L/P/./l_2/max$	planealg::L_l2_max_MlogM
L_l2_max_M2logM	$1L/P/./l_2/max$	planealg::L_l2_max_M2logM

L_lp_max_M4	$1L/P/./l_p/max$	planealg::L_lp_max_M4
linf_max	$1/P/./l_\infty/max$	planealg::linf_max
linf_v1_max	$1/P/v_i = 1/l_\infty/max$	planealg::linf_v1_max
in_convex_linf_max	$1/P/R = convex/l_\infty/max$	planealg::linf_max
N_linf_max	$N/P/./l_\infty/max$	planealg::N_linf_max
L_linf_max_M4	$1L/P/./l_\infty/max$	planealg::L_linf_max_M4
gauge_center	$1/P/./\gamma/\max$	gaugealg::max
L_block_max_M3	$1L/P/./\gamma_B/\max$	gaugealg::L_block_max_M3

Table 7.2: Planar algorithms of text-based LOLA

argument of -a	problem class	LOLA method
dir_median	$1/G_D/./d(V,V)/\sum$	lgraphalg::median
undir_median	$1/G/./d(V,V)/\sum$	lgraphalg::median
abs_undir_median	$1/G/./d(V,G)/\sum$	lgraphalg::abs_median
loc_tree_median	$1/T/./d(V,V)/\sum$	lgraphalg::loc_tree_median
abs_tree_median	$1/T/./d(V,T)/\sum$	lgraphalg::abs_tree_median
medPareto	$1/G/./d(V,G)/2 - \sum -par$	lgraphalg::medPareto_bi
	$1/G/./d(V,G)/Q - \sum -par$	lgraphalg::medPareto_Q
loc_medPareto	$1/G/./d(V,V)/Q - \sum -par$	lgraphalg::loc_medPareto
dir_medPareto	$1/G_D/./d(V,G)/Q - \sum -par$	lgraphalg::medPareto
dir_loc_medPareto	$1/G_D/./d(V,V)/Q - \sum -par$	lgraphalg::loc_medPareto
medLexi	$1/G/./d(V,G)/Q - \sum -lex$	lgraphalg::medLexi
loc_medLexi	$1/G/./d(V,V)/Q - \sum -lex$	lgraphalg::loc_medLexi
dir_medLexi	$1/G/./d(V,G)/Q - \sum -lex$	lgraphalg::medLexi
dir_loc_medLexi	$1/G/./d(V,V)/Q - \sum -lex$	lgraphalg::loc_medLexi
abs_dir_center	$1/G_D/./d(V,G)/\max$	lgraphalg::abs_center
loc_dir_center	$1/G_D/./d(V,V)/\max$	lgraphalg::center
loc_undir_center	$1/G/./d(V,V)/\max$	lgraphalg::center
abs_undir_center	$1/G/./d(V,G)/\max$	lgraphalg::abs_center
loc_tree_center	$1/T/./d(V,V)/\max$	lgraphalg::loc_tree_center
abs_tree_center	$1/T/./d(V,T)/\max$	lgraphalg::abs_tree_center
loc_cenPareto	$1/G/./d(V,V)/\max$	lgraphalg::loc_cenPareto
dir_loc_cenPareto	$1/G_D/./d(V,V)/\max$	lgraphalg::loc_cenPareto
dir_cenPareto	$1/G_D/./d(V,G)/\max$	lgraphalg::cenPareto
loc_cenLexi	$1/G/./d(V,V)/Q - \max -lex$	lgraphalg::loc_cenLexi
dir_loc_cenLexi	$1/G/./d(V,V)/Q - \max -lex$	lgraphalg::loc_medLexi
N_median_cplex	$N/G/./d(V,V)/\sum$	lgraphalg::N_median_cplex
N_median_partitioning	$N/G/./d(V,V)/\sum$	lgraphalg::N_median_partitioning
N_median_exchange	$N/G/./d(V,V)/\sum$	lgraphalg::N_median_exchange
N_median_greedy	$N/G/./d(V,V)/\sum$	lgraphalg::N_median_greedy
N_center_partitioning	$N/G/./d(V,V)/\max$	lgraphalg::N_center_partitioning

`N_center_greedy`	$N/G/./d(V,V)/\max$	`lgraphalg::N_center_greedy`

Table 7.3: Graph algorithms of text-based LOLA

Each algorithm of LOLA has been assigned a short, descriptive name which is the argument to the `-a` option. The choice of an algorithm determines the type of input files needed for the solution of the location problem. Table 7.1 on page 40 lists all feasible algorithm/data combinations, where "X" means *mandatory* and "O" means *optional*. Tables 7.2 and 7.3 show which class of location problems is solved by each algorithm and which LOLA-class method is used.

Examples A command could be

lola -lmydata.loc -al1_sum for a 1-location problem which has to be solved with respect to the l_1 norm and as a median problem.
or *lola -lmydata2.loc -aN_l1_max -mmat.1* for a N-location problem which has to be solved with respect to the l_1 norm and as a center problem.

7.3 Graphical Editor

This graphical editor enables the user to generate input for LOLA. The user can use the mouse to create a problem file for a location or a network problem. It will create a location file, a graph file, a gauge file or a restriction file in the correct input format. The menubar allows the user to choose File, Options or Help.

7.3.1 File

Under the File menu the following options are available.

New to clear the window.

Load to load a location or a restriction file (must have correct input format).

Load Interactionmatrix to load a matrix file for an N-location problem.

Save Location to save the coordinates and weights of locations.

Save Graph to save the coordinates and weights of nodes and edges.

Save Restriction to save the coordinates of restrictions.

Save Interactionmatrix to save a matrix file (available for number > 1).

Print to print the input as a postscript.

Quit to quit the Grapheditor and go back to LOLA.

7.3.2 Options

NewMax to change the *xmax* and *ymax* for the input window (default value is 50 and integer).

Draw Hull to draw the convex hull for all locations.

Number to insert the number of new locations. The user is able to insert the interdependence-matrix (default value is 1).

Locations

- clicking on the left mouse button defines an existing facility at the actual position.
- with pressed Shift-Key, clicking on the left mouse button on an existing point gives a dialog to change the weight of this point (default value is 1).
- with pressed Control-Key, clicking on the left mouse button on an existing point deletes this point.
- clicking on the right mouse button and moving the mouse moves the point until the mouse button is released.
- with pressed Shift-Key, clicking on the right mouse button on an existing point gives a dialog to insert a string as a description for this location.

Restrictions choose outside and the forbidden region is outside the restriction, otherwise the forbidden region is inside (default) and then choose one of the following restrictions.

Polyhedron

- clicking on the left mouse button defines a point of the polyhedron.
- double clicking on the left mouse button defines the last point of this polyhedron.
- with pressed Control-Key, clicking on the left mouse button on an existing point deletes all polyhedra.
- clicking on the right mouse button and moving the mouse moves the polyhedron until the mouse button is released.

Convex Polyhedron

- clicking on the left mouse button defines a point of the polyhedron.
- double clicking on the left mouse button defines the last point of this convex polyhedron and creates a convex polyhedron.
- with pressed Control-Key, clicking on the left mouse button on an existing point deletes all convex polyhedra.

Circle

- clicking on the left mouse button defines the middle- point of the circle.

- with pressed Shift-Key, clicking on the left mouse button and moving the mouse creates the circle and defines the radius if the mouse button is released.
- with pressed Control-Key, clicking on the left mouse button on an existing middlepoint deletes the circle.
- clicking on the right mouse button and moving the mouse moves the circle until the mouse button is released.

$\boxed{\text{Rectangle}}$

- clicking on the left mouse button defines the point on the left bottom of the rectangle.
- with pressed Shift-Key, clicking on the left mouse button and moving the mouse creates the rectangle and defines the sides a and b. If the mouse button is released then an angle α can be inserted to rotate the rectangle ($-90 < \alpha < 90$).
- with pressed Control-Key, clicking on the left mouse button on an existing point deletes the rectangle.
- clicking on the right mouse button and moving the mouse moves the rectangle until the mouse button is released.

$\boxed{\text{Gauge}}$ to choose the maximal extension of the gauge and to create gauges with the buttons:

- $\boxed{\text{Save Gauge}}$ to save single gauge in a file.
- $\boxed{\text{Show/Hide Unit Ball}}$ to make the unit ball of the gauge (in)visible.
- $\boxed{\text{Clear}}$ to clear the input window.
- $\boxed{\text{Append to Gauge-List}}$ to append the actual gauge to a polygauge list (if no such list exists, a new one is opened).
- $\boxed{\text{Save Gauge-List}}$ to save the whole polygauge list in a file.
- $\boxed{\text{Clear Gauge-List}}$ to delete the existing polygauge list.
- $\boxed{\text{Symmetr/Unsymmetr}}$ to create symmetrical or unsymmetrical polygauges (default is symmetrical: Only one of two symmetrical extreme points of the unit ball has to be added, the other one is added automatically).
- $\boxed{\text{Refresh}}$ to refresh the input window.
- $\boxed{\text{Close}}$ to close the input window.

$\boxed{\text{Undirected Graph}}$ to choose whether the graph is a directed one or an undirected one.

- to draw an edge click with the middle mouse button (assuming the mouse has three buttons) on one location and release the button on another location.
- in case the mouse has two buttons only, the user can use the Alt-Key together with the left mouse button to draw edges.
- to insert an edge-weight press the Shift-Key and click on the left mouse button.
- to delete an edge press the Control-Key and click on the left mouse button.

In the bottom appear the actual coordinates of the mouseposition and the actual insert Mode.

7.4 Other Software Used by LOLA

Tools and utilities which are used:

- LEDA,
- TCL/TK,
- CPLEX, or some other type of LP–solver, which is able to process files in the MPS-Format

The implementation language is C/C++.

Chapter 8

System Design

The system consists of the three main classes *planealg* (P), *graphalg* (G) and *discalg* (D). Through these classes all routines in the library can be called and controlled.

8.1 System Structure

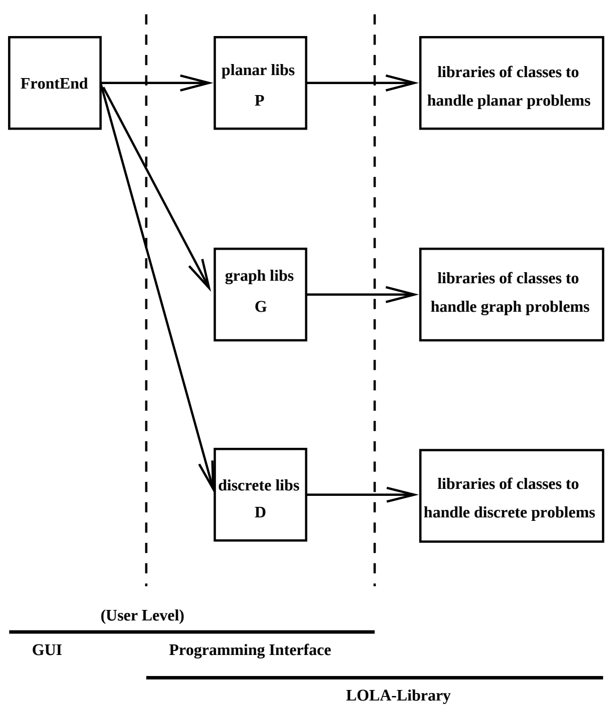

Figure 8.1: System Structure. The layers of the user–interfaces and the libraries are shown. The internal design of the libraries for handling the three types of location problems is specified in the text.

8.2 Component Structure Type P

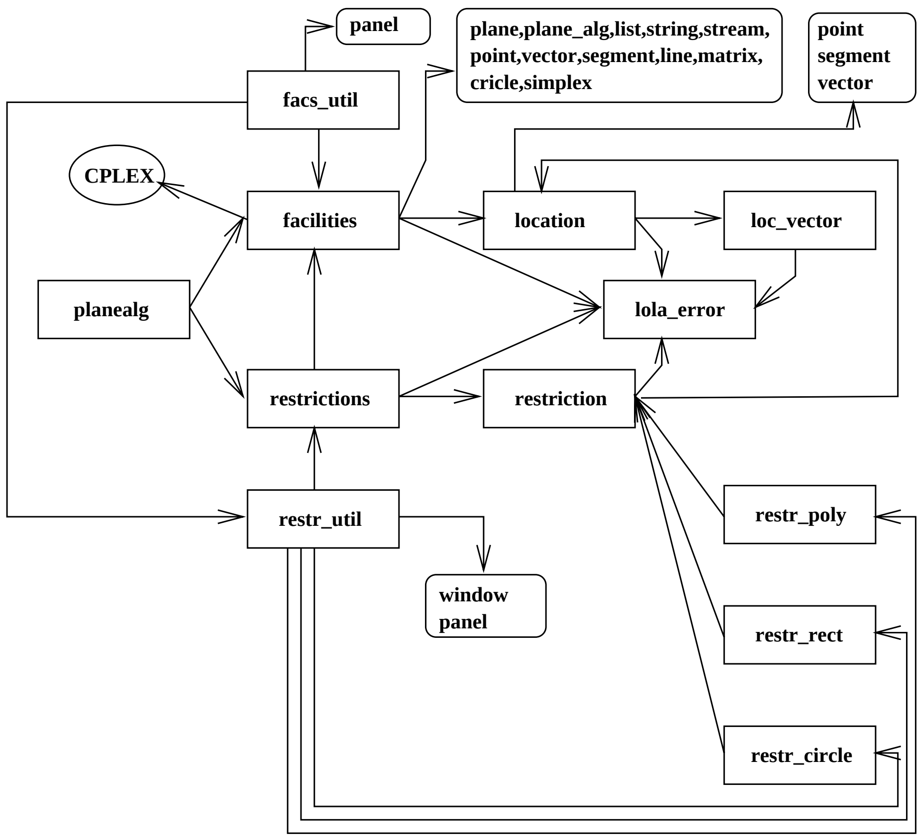

Figure 8.2: Component Structure P. The arrows stand for “uses/accesses”–relations between the modules. Components of LOLA are represented by rectangles, components of LEDA are represented by smooth rectangles, and other components are represented by ellipsoids.

This is the graphical representation of the inner structure of the Type P–libraries of LOLA. All dependencies and accesses to LEDA and other pre–requisites are shown in this figure. Furthermore the coupling of this part of the system is described by the arrows.

8.3 Library Inclusions Type P

Figure 8.3 shows the LOLA classes and the corresponding libraries in which they are located. The structure provides a separation of unrestricted problems (libLp), restricted problems (libLpr) and utility functions (libLpu).

If an executable program is statically linked with the LOLA libraries, the library sequence has to be libLpu, libLpr, libLp.

Example (with GNU’s C++):

g++ -o *myprog myprog.o* -l*otherlibs* -l*Lpu* -l*Lpr* -l*Lp* -l*otherlibs*

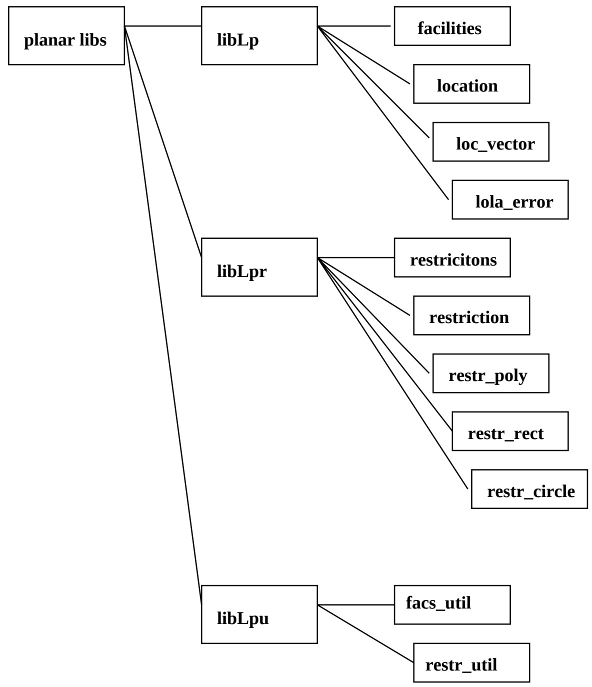

Figure 8.3: Library Structure P. Showing the inclusions of classes to libraries.

This will produce the statically linked executable file *myprog*. For more information about the linker see your GNU C/C++ manual or your system's C/C++ compiler manual.

Part II

The LOLA Libraries

Chapter 9

Unrestricted Planar Classes

9.1 Real-Valued Location-Vectors (Loc_Vector)

1. Definition

An instance of the data type Loc_Vector consists of an n-dimensional array of reals.

2. Creation

$Loc\_Vector \quad V(int\ dim = 0);$

creates a dim-dimensional Loc_Vector, default dimension is 0.

$Loc\_Vector \quad V(double\ dim, ...);$

creates an instance Loc_Vector with the following initialization: $v(dim, arg_1, .., arg_{dim}.)$. All numbers must be doubles.

3. Operations

Loc_Vector	$V\ +\ \&v1$	returns the sum of V and $v1$.
Loc_Vector	$V\ -\ \&v1$	returns the result of the subtraction of $v1$ from V.
$Loc\_Vector\&$	$V\ += \ \&v1$	adds $v1$ to V.
$Loc\_Vector\&$	$V\ -= \ \&v1$	subtracts $v1$ from V.
$Loc\_Vector\&$	$V\ *= \ double\ scalar$	multiplies V by $scalar$.

Loc_Vector&	$V \ += \ double\ scalar$	adds *scalar* to V.
Loc_Vector&	$V \ -= \ double\ scalar$	subtracts *scalar* from V.
Loc_Vector&	$V \ /= \ double\ scalar$	divides V by *scalar*.
Loc_Vector	$V \ * \ double\ scalar$	returns the product of V and *scalar*.
Loc_Vector	$V \ + \ double\ scalar$	returns the sum of V and *scalar*.
Loc_Vector	$V \ - \ double\ scalar$	returns the result of the subtraction of *scalar* from V.
Loc_Vector	$V \ / \ double\ scalar$	returns the result of the division of V by *scalar*.
double	$V * V_2$	inner product of V with V_2.
double&	$V\ [int\ idx]$	returns a reference to the *idx*-th component of V.
bool	$V \ < \ \&v1$	returns true if $V < v1$ for all elements.
bool	$V \ <= \ \&v1$	returns true if $V \leq v1$ for all elements.
bool	$V \ > \ \&v1$	returns true if $V > v1$ for all elements.
bool	$V \ >= \ \&v1$	returns true if $V \geq v1$ for all elements.
bool	$V \ != \ \&v1$	returns true if $V \neq v1$ for all elements.
bool	$V \ == \ \&v1$	returns true if $V = v1$ for all elements.
Loc_Vector&	$V = V_2$	assignment operator.
double	V.norm(*int p*)	returns the p-norm (p-metric) of V.
void	V.abs()	returns the absolute value of V.
int	V.size()	returns the dimension of V.
bool	V.null_chk()	returns true if one or more components of V are zero.
Loc_Vector	V.rotate(*double phi*)	rotates V by *phi*.

ostream&	<< (ostream &stream, Loc_Vector &v)	writes V componentwise.

4. Implementation

All operations on a Loc_Vector take time $O(size())$.

9.2 Real-Valued Locations with weights (location)

1. Definition

An instance of the data type *location* consists of two Loc_Vectors.

2. Creation

location $L(int\ dim = 2,\ int\ wht = 1);$

creates a location with a dim-dimensional Loc_Vector for the coordinates and a wht-dimensional Loc_Vector for the weights.

location $L(double\ dim, ...);$

creates an instance *Location* with the following initialization: $v(dimension, \#weights, coord1, coord2, ..., wht1, wht2, ...)$.
All numbers must be doubles.

3. Operations

double	L.norm($int\ p$)	returns the p-norm (p-metric) of L.
int	L.dim()	returns the dimension of L.
int	L.wht_dim()	returns the number of weights of L.
int	L.overall()	returns $dim() + wht\_dim()$ of L.
$Loc\_Vector\&$	L.pos()	returns a reference to the coordinate Loc_Vector.
$Loc\_Vector\&$	L.wht()	returns a reference to the weights Loc_Vector.
void	L.transform()	transforms the coordinates. Precondition: $L.dim() = 2$.

void	L.retransform()	retransforms the coordinates. Precondition: $L.dim() = 2$.
double	L.r_angle(*location* &$L1$)	calculates the radian angle between L and $L1$.
double	L.d_angle(*location* &$L1$)	calculates the degree angle between L and $L1$.
point	L.loc2point()	transforms *location* L into type *point*. $Precondition: dim() = 2$.
vector	L.loc2vector()	transforms *location* L into type *vector* (not Loc_Vector). $Precondition: dim() > 0$.
segment	L.makesegment(*location* $L1$)	returns a *segment* built of *location* L and $L1$. $Precondition: L.dim() = 2$.
double&	L [*int* i]	returns a reference to the i-th coordinate of L.
double&	L (*int* j)	returns a reference to the j-th weight of L.
double	L $*$ &$L1$	returns product of L and $L1$.
location	L $*$ *double* &*scalar*	returns product of L with *scalar*.
location	L / *double* &*scalar*	returns the result of the division of L by *scalar*.
location	L + &$L1$	returns *location* which coordinates are the sum of of the $L1$-coordinates and the L-coordinates.
location	L − &$L1$	returns *location* which coordinates are the difference of of the $L1$-coordinates and the L-coordinates.
int	L == &$L1$	tests for equality in coordinates *and* weights of L and $L1$ (true = 1, false = 0).
int	L != &$L1$	tests for inequality in coordinates *and* weights of L and $L1$ (true = 1, false = 0).
location&	L.=(*location* &$L1$)	assignment operator.

9.3 Line (Line)

1. Definition

An instance of the data type *Line* consists of three doubles and the line-defining locations; $(ax + by - c = 0)$.

2. Creation

Line V;

creates a zero-*Line*.

Line V(*location* $l1$, *location* $l2$);

creates a *line* defined by $l1$ and $l2$.

3. Operations

Line&	V = &L	assignment operator.
double&	V.m()	returns a reference to the slope M of V.
location&	V.L1()	returns a reference to the first defining location $L1$.
location&	V.L2()	returns a reference to the first defining location $L2$.
double&	V.a()	returns a reference to the x-coefficient of V.
double&	V.b()	returns a reference to the y-coefficient of V.
double&	V.c()	returns a reference to the constant c.
bool	V.parallel(*Line* &L)	returns true if the lines V and L are parallel; otherwise false.
double	V.weighted_distance(*location* &L)	calculates the weighted euclidean distance from L to the line V.
double	V.distance(*location* &L)	calculates the unweighted euclidean distance from L to the line V.
double	V.lp_distance(*location* &L, *int* p)	

		calculates the unweighted l_p-distance from L to the line V.
location	V.intersection(*Line* &L)	calculates the intersection-location of the two lines.
bool	V < &L	returns true if L is parallel to V and lies below V.
bool	V > &L	returns true if L is parallel to V and lies above V.
bool	V == &L	returns true if the lines are identical.

9.4 1-line-algorithms (lines)

1. Definition

An instance of the data type *lines* consists of 1-line algorithms.

2. Median-problems

double V.L_l2_sum_M3(*facilities*& *facs*)

computes the optimum for problem class $1L/P/./l_2/\sum$ and returns the objective value.

double V.L_l1_sum_M3(*facilities*& *facs*)

computes the optimum for problem class $1L/P/./l_1/\sum$ and returns the objective value.

double V.L_linf_sum_M3(*facilities*& *facs*)

computes the optimum for problem class $1L/P/./l_\infty/\sum$ and returns the objective value.

double V.L_lp_sum_M3(*facilities*& *facs*, *int p*)

computes the optimum for problem class $1L/P/./l_p/\sum$ and returns the objective value.

double V.L_l2_sum_M2logM(*facilities*& *facs*)

computes the optimum for problem class $1L/P/./l_2/\sum$ and returns the objective value.

double V.L_l2_sum_M2($facilities\&\ facs$)

computes the optimum for problem class $1L/P/v_i = 1/l_2/\sum$ and returns the objective value.

double V.L_block_sum_M3($list$<$polygauge$> $\&LG, list$<int> $\&Ln,\ facilities\&\ facs$)

computes the optimum for problem class $1L/P/./l_{block}/\sum$ and returns the objective value.

3. Center-problems

double V.L_l2_max_M4($facilities\&\ facs$)

computes the optimum for problem class $1L/P/./l_2/max$ and returns the objective value.

double V.L_l1_max_M4($facilities\&\ facs$)

computes the optimum for problem class $1L/P/./l_1/max$ and returns the objective value.

double V.L_linf_max_M4($facilities\&\ facs$)

computes the optimum for problem class $1L/P/./l_\infty/max$ and returns the objective value.

double V.L_l2_max_MlogM($facilities\&\ facs$)

computes the optimum for problem class $1L/P/v_i = 1/l_2/max$ and returns the objective value.

double V.L_l2_max_M2logM($facilities\&\ facs$)

computes the optimum for problem class $1L/P/./l_2/max$ and returns the objective value.

double V.L_lp_max_M4($facilities\&\ facs,\ int\ p$)

computes the optimum for problem class $1L/P/./l_p/max$ and returns the objective value.

double V.L_block_max_M4($list$<$polygauge$> $\&LG, list$<int> $\&Ln,\ facilities\&\ facs$)

computes the optimum for problem class $1L/P/./l_{block}/max$ and returns the objective value.

4. Useful functions

double V.lp_distance_2_line(*location* &$l1$, *location* &$l2$, *location* &L, *int* p)

calculates the l_p-distance of L to the line defined by $l1$ and $l2$.

double V.objective_function_l2(*location* **locs*, *int start*, *int* N, *location* &$l1$, *location* &$l2$)

calculates the value of the objective function with metric l_2 for the line defined by $l1$ and $l2$ and for the locations with *start* as the beginning index (first element has to be 1) and N as the index for the final element.

void V.max_line_l2(*location* &$l1$, *location* &$l2$, *location* &$l3$, *location* &$r1$, *location* &$r2$)

calculates the optimal l_2-line defined by $r1$ and $r2$ for $l1$, $l2$ and $l3$.

double V.objective_function_linf(*location* **locs*, *int start*, *int* N, *location* &$result1$, *location* &$result2$)

calculates the value of the objective function with metric l_∞ for the line defined by $l1$ and $l2$ and for the locations with *start* as the beginning index (first element has to be 1) and N as the index for the final element.

double V.objective_function_l1(*location* **locs*, *int start*, *int* N, *location* &$result1$, *location* &$result2$)

calculates the value of the objective function with metric l_1 for the line defined by $l1$ and $l2$ and for the locations with *start* as the beginning index (first element has to be 1) and N as the index for the final element.

void V.max_line_l1(*location* &$l1$, *location* &$l2$, *location* &$l3$, *location* &$r1$, *location* &$r2$)

calculates the optimal l_1-line defined by $r1$ and $r2$ for $l1$, $l2$ and $l3$.

double V.objective_function_lp(*location* **locs*, *int start*, *int* N, *location* &$result1$, *location* &$result2$, *int*

calculates the value of the objective function with metric l_p for the line defined by $l1$ and $l2$ and for the locations with *start* as the beginning index (first element has to be 1) and N as the index for the final element.

void V.max_line_lp(*location* &$l1$, *location* &$l2$, *location* &$l3$, *location* &$r1$, *location* &$r2$, *int* p)

calculates the optimal l_p-line defined by $r1$ and $r2$ for $l1$, $l2$ and $l3$.

double V.objective_function_block(*location* ∗*locs*, *int start*, *int N*, *location* &*l*1, *location* &*l*2, *Blocknorm*

calculates the value of the objective function with block-metric for the line defined by $l1$ and $l2$ and for the locations with *start* as the beginning index (first element has to be 1) and N as the index for the final element.

void V.max_line_block(*location* &*l*1, *location* &*l*2, *location* &*l*3, *location* &*r*1, *location* &*r*2, *Blocknorm*

calculates the optimal *block*-line defined by $r1$ and $r2$ for $l1$, $l2$ and $l3$.

int V.geometric_duality(*location* ∗*locs*, *location* &*pivot*, *int M*)

sorts all M locations *locs* with respect to *pivot* using geometric duality.

void V.y_quicksort(*location* ∗*tosort*, *int l*, *int r*, *location* ∗*old*)

sorts the locations *tosort* from l (usual 0) to r (usual number of locations minus 1) dependent on the y-coordinate (all locations have to be 2-dimensional) using a recursive quicksort-algorithm (*old* should be the last location).

5. Implementation

The time needed by the algorithms is given in the name of each, e.g. algorithm $L\_l2\_maxM2\log M$ takes $o(M^2 \log M)$ time to run.

9.5 Facilities (facilities)

1. Definition

An instance of the data type *facilities* consists of a list of locations.

2. Creation

facilities EX; introduces a variable EX of type *facilities*.

3. Operations

3.1 Input and Output

string	EX.name(*int location*)	provides a name for *location*.
void	EX.set_name(*int location*, *string* &*name*)	specifies a name for *location*.
void	EX.Write(*ostream* &*out*)	prints all facilities.
void	EX.WriteOpt(*ostream* &*out*)	prints the set of optimal solutions.

3.2 Operations

bool	EX.element(*location* &*Loc*)	returns true if *Loc* is an element of EX, else false.
location	EX.get_element(*int idx*)	returns the *idx*-element of the EX.
int	EX.get_idx(*location* &L)	returns the index of L otherwise the total number of locations.
void	EX.remove(*location* &L)	removes L from EX if possible.
void	EX.remove(*int idx*)	removes the *idx*-element of EX if possible.
double	EX.max_direction_value(*int* i)	returns the maximum value of coordinate i of all locations of EX.
double	EX.min_direction_value(*int* i)	returns the minimum value of coordinate i of all locations of EX.
double	EX.diff_value(*int* i)	returns the maximal difference between coordinates i of any two locations of EX.

3.3 Algorithms

3.3.1 Median-problems

double	EX.l1_sum(*int weight*, *bool erase*)	computes the optimum for problem class $1/P/./l_1/\sum$ and returns the objective value.

double EX.l2sqr_sum($int\ weight, bool\ erase$)

computes the optimum for problem class $1/P/./l_2^2/\sum$ and returns the objective value.

double EX.linf_sum($int\ weight, bool\ erase$)

computes the optimum for problem class $1/P/./l_\infty/\sum$ and returns the objective value.

double EX.l2_sum($int\ weight, double\ epsilon, bool\ erase$)

computes the optimum for problem class $1/P/./l_2/\sum$ (Weiszfeld-Algorithm) and returns the objective value.

double EX.Weiszfeld($double\ epsilon$) calls l2_sum(0, epsilon, true).

double EX.lp_sum($int\ p, double\ epsilon, double\ delta, int\ iter\_max, int\ weight, bool\ erase$)

computes the optimum for problem class $1/P/./l_p/\sum$ (Generalized Weiszfeld-Algorithm) and returns the objective value.

double EX.l2sqr_qsum($bool\ erase$)

computes the optimum for problem class $1/P/./l_2^2/Q-\sum$ (Q-criterial problem) and returns the objective value.

double EX.l1_2sum($bool\ erase$)

computes the optimum for problem class $1/P/./l_1/2-\sum$ (bi-criterial problem) and returns the objective value.

double EX.linf_2sum($bool\ erase$)

computes the optimum for problem class $1/P/./l_\infty/2-\sum$ (bi-criterial problem) and returns the objective value.

double EX.N_l1_sum($int\ N, matrix\&\ w, bool\ erase$)

computes the optimum for problem class $N/P/./l_1/\sum$ with matrix of interdependencies w and returns the objective value.

double EX.N_linf_sum($int\ N, matrix\&\ w, bool\ erase$)

computes the optimum for problem class $N/P/./l_\infty/\sum$ with matrix of interdependencies w and returns the objective value.

double	EX.N_l2sqr_sum(*int* N, *matrix*& w, *bool* *erase*)	
		computes the optimum for problem class $N/P/./l_2^2/\sum$ with matrix of interdependencies w and returns the objective value.
double	EX.N_l2sqr_sum(*int* N, *matrix*& w, *bool* *erase*)	
		computes the optimum for problem class $N/P/./l_p/\sum$ (Version 1) and returns the objective value.
double	EX.N_l2sqr_sum(*int* N, *matrix*& w, *bool* *erase*)	
		computes the optimum for problem class $N/P/./l_p/\sum$ (Version 2) and returns the objective value.
double	EX.fn_sum(*int* n, *int* p)	computes objective value of X_n regarding all locations of EX.
double	EX.fex_sum(*int* p)	computes objective value of all solutions regarding all facilities.
double	EX.fnew_sum(*matrix*& w, *int* p)	
		computes objective value of X_n's to each other.
void	EX.N_l1_sum_gen_LP(*int* N, *matrix*& w, *char* $*$ *lp*1, *char* $*$ *lp*2)	
		generates LP for problem class $N/P/./l_1/\sum$.
void	EX.write_mps(*vector*& OBJ, *matrix*& $CONSTRAINTS$, *vector*& RHS, *list*<*char*>& EQU, *cha* *filename*)	

writes down a given objective function vector OBJ, matrix of constraints $CONSTRAINTS$, right hand side vector RHS, (in)equality sign list EQU to file $filename$ in mps–format. For each row of $CONSTRAINTS$ there must be a corresponding entry in EQU of the set E(for equal to), L(for less than or equal to), G(for greater than or equal to), and N(for neutral). Remember that the objective row must not be mentioned in EQU. EQU is only for use with the $CONSTRAINT$–rows.

double EX.gauge_sum(*list<polygauge>& LG, list<int>& Ln, bool erase*)

computes the optimum for problem class $1/P/./\gamma/\sum$.

double EX.gauge_bi_crit_sum(*list<polygauge>& LG, list<int>& Ln, bool erase*)

computes the optimum for problem class $1/P/./\gamma/2-\sum_{par}$ and returns the number of situations (line segments and cells) the pareto set consists of.

3.3.2 Center-problems

double EX.linf_max($bool\ erase = true$)

computes the optimum for problem class $1/P/./l_\infty/max$ and returns the objective value.

double EX.l1_max($bool\ erase = true$)

computes the optimum for problem class $1/P/./l_1/max$ and returns the objective value.

double EX.l1_v1_max($bool\ erase = true$)

computes the optimum for problem class $1/P/w_m = 1/l_1/max$ and returns the objective value.

double EX.l2_max($bool\ erase = true$)

computes the optimum for problem class $1/P/./l_2/max$ and returns the objective value.

double EX.linf_v1_max(*bool erase* = *true*)

computes the optimum for problem class $1/P/./l_\infty/max$ and returns the objective value.

double EX.elz_hearn(*bool erase* = *true*)

computes the optimum for Elzinga-Hearn-Algorithm and returns the objective value.

double EX.N_linf_max(*int N*, *matrix& W*, *bool erase* = *true*)

computes the optimum for problem class $N/P/./l_\infty/max$ and returns the objective value.

double EX.N_l1_max(*int N*, *matrix& W*, *bool erase* = *true*)

computes the optimum for problem class $N/P/./l_1/max$ and returns the objective value.

double EX.N_linf_max_mps(*int N*, *matrix& W*, *bool erase* = *true*)

computes the optimum for problem class $N/P/./l_\infty/max$ using *mps*-files and CPLEX and returns the objective value. CPLEX generates also two files named "sol1.txt" and "sol2.txt".

double EX.N_l1_max_mps(*int N*, *matrix& W*, *bool erase* = *true*)

computes the optimum for problem class $N/P/./l_1/max$ using *mps*-files and CPLEX and returns the objective value. CPLEX generates also two files named "sol1.txt" and "sol2.txt".

3.4 Useful Functions

double EX.objective_value_computation($list<polygauge>\&\, Lg, list<int>\&\, Ln, point\, X,\ int\, q$)

computes the objective value for a gauge problem.

double EX.objective_value_computation($list<polygauge>\&\, Lg, list<int>\&\, Ln, point\, X,\ int\, q$)

computes the subgradient value for a gauge problem.

list<segment> EX.all_segments_computation(*list<polygauge>*& Lg, *list<int>*& Ln, *point* Y, *double* f_X, *int* q)

computes all segments for all locations for 1-criterial.

list<segment> EX.all_segments_computation(*list<polygauge>*& Lg, *list<int>*& Ln, *point* Y, *double* f_X, *int* q)

computes the function values for the grid-points.

vector EX.niveauline_vector_computation(*list<polygauge>*& Lg, *list<int>*& Ln, *point* X, *int* q)

computes the direction vector for level curve.

list<point> EX.computation_of_lex_solutions(*list<polygauge>*& Lg, *list<int>*& Ln, *point* X, *int* $q1$, *int* $q2$, G

computes the lexicographical solution.

list<location> EX.non_trivial_case(*list<polygauge>*& Lg, *list<int>*& Ln, *list<point>*& Lex_sol_12, *list<point>*&

computes the pareto solution for a non trivial case.

9.6 Facilities-utilities (facs_util)

1. Definition

An instance EX of the data type *facs_util* supports the data type *facilities*. It provides some useful routines to handle the class *facilities*.

2. Creation

facs_util EX ; creates a list of location EX of type *facs_util*.

3. Operations

3.1 Input and Output

void EX.ReadLoc(*ifstream*& *file*) reads all locations from *file*.

void EX.ReadLoc_Res(*ifstream*& *file*) reads all locations with results from *file*.

void EX.ReadLoc(*ifstream*& *file*, *list<int>*& *Dist*) reads all locations from *file* and returns a list to associate distance functions.

void *EX*.ReadRestr(*ifstream*& *file*) reads all restrictions from *file*.

void *EX*.SaveLoc(*ostream*& *file*) saves all locations into *file*.

void *EX*.SaveLoc_Res(*ostream*& *file*, *double objval*, *string normact*)

saves all locations and the results of the current problem into *file*.

matrix *EX*.ReadMat(*ifstream*& *file*) reads a matrix from *file* for a *multi* − *facilities* − *problem*.

void *EX*.View(*double objval*, *string normact*, *list*<*location*>& *AlgSol*, *restrictions*& *Restr*)

shows the results and locations of the current problem.

void *EX*.View(*double objval*, *string normact*, *list*<*location*>& *AlgSol*, *restrictions*& *Restr*, *list*<*polygauge*>

shows the results and locations of the current problem.

void *EX*.View(*double objval*, *string normact*, *list*<*location*>& *AlgSol*, *restrictions*& *Restr*, *list*<*polygon*>&

shows the results and locations of the current problem.

void *EX*.View(*double objval*, *string normact*, *list*<*location*>& *AlgSol*, *restrictions*& *Restr*, *list*<*polygon*>&

shows the results and locations of the current problem.

Chapter 10

Restricted Planar Classes

10.1 Restriction (restriction)

1. Definition

Restriction is one of three restriction-types (*restr_poly*, *restr_circle* or *restr_rect*) and has the following virtual function for all of them.

2. Creation

restriction R; creates a variable R of type *restriction*.

3. Operations

int	type()	returns the type of the restriction (*restr_p*, *restr_c* or *restr_r*).
bool	inout()	returns true if the forbidden region should be inside R, false otherwise.
void	in_forbid()	changes forbidden region to inside.
void	out_forbid()	changes forbidden region to outside.
bool	inside(point p)	returns true if p lies inside R, false otherwise.
bool	inside(location Loc)	returns true if *Loc* lies inside R, false otherwise.
bool	inside(segment seg)	returns true if *seg* lies inside R, false otherwise.

list<point>	intersect(ray r)	returns $R \cap r$ as a list of points.
list<point>	intersection(line l)	returns $R \cap l$ as a list of points.
list<point>	intersection(segment s)	returns $R \cap s$ as a list of points.
list<point>	intersection(polygon P)	returns $R \cap P$ as a list of points.
list<point>	intersection(circle C)	returns $R \cap C$ as a list of points.
list<location>	proj_l2sqr(location opt)	returns the projected locations for the l_2^2 Norm.
double	proj_v1_l2_max(location opt, facilities facs, double z_opt)	returns the objective value for the projected solution in opt for the problem class $1/P/w_m = 1/l_2/max$.

Non-virtual functions

bool	inside_all(restriction∗ restrict, list<location> opt)	returns true if all locations lie inside the restriction, false otherwise.
restriction∗	inside_which(list<restriction∗ > restrict, location loc)	returns *restriction* where *loc* lies inside.
restriction∗	inside_which(list<restriction∗ > restrict, segment seg);	returns *restriction* where *seg* lies inside.
restriction∗	inside_which(list<restriction∗ > restrict, list<location> opt);	returns *restriction* where all locations from *opt* lie inside.

10.2 Polygon as a restriction (restr_poly)

1. Definition

An instance P of the data type *restr_poly* is a simple polygon in the two-dimensional plane defined by the sequence of its vertices. The number of vertices is called the size of P. A *restr_poly* with empty vertex sequence is called empty.

2. Creation

restr_poly P;

introduces a variable P of type *restr_poly*. P is initialized to the empty *restr_poly*.

restr_poly P(*list<point>* p);

introduces a variable P of type *restr_poly*. P is initialized to the *restr_poly* with vertex sequence p.
Precondition: The vertices in p define a simple polygon.

3. Operations

list<point>	P.vertices()	returns the sequence of vertices of P in counterclockwise ordering.

list<segment>	P.segments()	returns the sequence of bounding segments of P in counterclockwise ordering.
bool	P.convex()	returns true if P is convex, false otherwise.
int	P.size()	returns the size of P.
bool	P.empty()	returns true if P is empty, false otherwise.
bool	P.==(*restr_poly* $P1$)	test for equality of P and $P1$.

10.3 Circle as a restriction (restr_circle)

1. Definition

An instance C of the data type *restr_circle* is a circle in the two-dimensional plane, i.e. the set of points having a certain distance r from a given point p. r is called the radius and p is called the center of C. The *restr_circle* with center $(0,0)$ and radius 0 is called the empty *restr_circle*.

2. Creation

restr_circle C;

introduces a variable C of type *restr_circle*. C is initialized to the empty *restr_circle*.

restr_circle C(*point* c, *double* r);

introduces a variable C of type *restr_circle*. C is initialized to the circle with center c and radius r.

restr_circle C(*double* x, *double* y, *double* r);

introduces a variable C of type *restr_circle*. C is initialized to the circle with center (x, y) and radius r.

restr_circle C(*point* a, *point* b, *point* c);

introduces a variable C of type *restr_circle*. C is initialized to the circle through points a, b, and c. *Precondition*: a, b, and c are not collinear.

3. Operations

point	C.center()	returns the center of C.
double	C.radius()	returns the radius of C.
double	C.distance(*point* p)	returns the distance between C and p (negative if p inside C).
bool	$C\ ==\ D$	tests for equality of C and D.

10.4 Rectangle as a restriction (restr_rect)

1. Definition

An instance R of the data type *restr_rect* is a simple rectangle in the two-dimensional plane defined by the sequence of its vertices. The number of vertices is called the size of R. A *restr_rect* with empty vertex sequence is called empty.

2. Creation

restr_rect $R(point\ p, double\ x, double\ y, double\ rad)$;

introduces a variable R of type *restr_rect*. R is initialized to the *restr_rect* with corner p and sides of length x and y and rotated by rad.

restr_rect $R(point\ p, double\ x, double\ y)$;

introduces a variable R of type *restr_rect*. R is initialized to the *restr_rect* with corner p and sides of length x and y.

restr_rect $R(point\ p1, point\ p2, point\ p3, point\ p4)$;

introduces a variable R of type *restr_rect*. R is initialized to the *restr_rect* with vertices $p1$, $p2$, $p3$, $p4$.

restr_rect $R(point\ p1, point\ p2)$;

introduces a variable R of type *restr_rect*. R is initialized to the *restr_rect* with opposite vertices $p1$ and $p2$.

3. Operations

double	R.area()	returns the area of R.
int	R.==(*restr_rect* $r1$)	tests for equality of R and $r1$.

10.5 Algorithms for Forbidden Regions (restrictions)

1. Definition

An instance of the data type *restrictions* consists of the algorithm to solve planar location problems with restrictions.

2. Creation

restrictions R; creates restrictions R

3. Operations

list<*location*> R.alg_solution() returns the solution for the current problem.

double R.l1_sum(*facilities*& EX, *bool* $erase$)

computes the optimum for problem class $1/P/R/l_1/\sum$ and returns the objective value.

double R.linf_sum(*facilities*& EX, *bool* $erase$)

computes the optimum for problem class $1/P/R/l_\infty/\sum$ and returns the objective value.

double R.l2sqr_sum(*facilities*& EX, *bool* $erase$)

computes the optimum for problem class $1/P/R/l_2^2/\sum$ and returns the objective value.

double R.lp_sum(*facilities*& EX, *int* p, *double* $epsilon$, *double* $delta$, *int* $iter\_max$, *bool* $erase$)

computes the optimum for problem class $1/P/R/l_p/\sum$ and returns the objective value.

double R.l2_v1_max(*facilities*& EX, *bool* $erase$)

computes the optimum for problem class $1/P/R = convex\,polyhedron, w_m = 1/l_1/\sum$ and returns the objective value.

double R.linf_max($facilities\&\ EX, bool\ erase$)

computes the optimum for problem class $1/P/R = convex\,restriction/l_\infty/\sum$ and returns the objective value.

double R.l1_max($facilities\ \&EX, bool\ erase$)

computes the optimum for problem class $1/P/R = convex\,restriction/l_1/\sum$ and returns the objective value.

double R.L_RkonvPoly_l2_sum($facilities\&\ EX,\ bool\ erase$)

computes the optimum for problem class $1L/P/R = convex\,polyhedron/l_2/\sum$ and returns the objective value.

double R.L_RkonvPoly_lp_sum($facilities\&\ EX,\ bool\ erase,\ int\ norm$)

computes the optimum for problem class $1L/P/R = convex\,polyhedron/l_p/\sum$ and returns the objective value.

double R.L_RkonvPoly_lp_sum($facilities\&\ EX,\ bool\ erase,\ int\ norm$)

computes the optimum for problem class $1L/P/R = convex\,polyhedron/l_1/\sum$ and returns the objective value.

double R.L_RkonvPoly_lp_sum($facilities\&\ EX,\ bool\ erase,\ int\ norm$)

computes the optimum for problem class $1L/P/R = convex\,polyhedron/l_\infty/\sum$ and returns the objective value.

double R.L_RkonvPoly_lp_sum($facilities\&\ EX,\ bool\ erase,\ int\ norm$)

computes the optimum for problem class $1L/P/R = convex\,polyhedron/\gamma_B/\sum$ and returns the objective value.

10.6 Polygon as a barrier (polygon_barrier)

1. Definition

An instance P of the data type *polygon_barrier* is a simple polygon in the two-dimensional plane defined by the sequence of its vertices. The number of vertices is called the size of P. A *polygon_barrier* with empty vertex sequence is called empty.

2. Creation

polygon_barrier P;

introduces a variable P of type *polygon_barrier*. P is initialized to the empty *polygon_barrier*.

polygon_barrier P(*list*<*point*> Pt);

introduces a variable P of type *polygon_barrier*. P is initialized to the polygon with vertex sequence Pt.
Precondition: The vertices in Pt define a simple polygon.

3. Operations

list<*point*>	P.vertices()	returns the sequence of vertices of P in counterclockwise ordering.
bool	P.visible(*point* &$p1$, *point* &$p2$)	returns true if $p2$ is visible from $p1$ with respect to P.

10.7 Circle as a barrier (circle_barrier)

1. Definition

An instance C of the data type *circle_barrier* is a circle in the two-dimensional plane, i.e. the set of points having a certain distance r from a given point p. r is called the radius and p is called the center of C. The *circle_barrier* with center $(0,0)$ and radius 0 is called the empty *circle_barrier*.

2. Creation

circle_barrier C;

introduces a variable C of type *circle_barrier*. C is initialized to the empty *circle_barrier*.

circle_barrier C(*point* p, *double* r);

introduces a variable C of type *circle_barrier*. C is initialized to the circle with center p and radius r.

circle_barrier C(*double* x, *double* y, *double* r);

introduces a variable C of type *circle_barrier*. C is initialized to the circle with center (x, y) and radius r.

circle_barrier C(*point* a, *point* b, *point* c);

introduces a variable C of type *circle_barrier*. C is initialized to the circle through points a, b, and c. *Precondition*: a, b, and c are not collinear.

3. Operations

point	C.center()	returns the center of C.
double	C.radius()	returns the radius of C.
bool	C.visible(*point* $p1$, *point* $p2$)	returns true if $p2$ is visible from $p1$ with respect to P.

10.8 Segment as a barrier (segment_barrier)

1. Definition

An instance S of the data type *segment_barrier* is a segemnt in the two-dimensional plane defined by its the points including the segment.

2. Creation

segment_barrier S;

introduces a variable S of type *segment_barrier*. S is initialized to the empty *segment_barrier*.

segment_barrier S(*point* $p1$, *point* $p2$);

introduces a variable S of type *segment_barrier*. S is initialized to the segment $(p1, p2)$.

3. Operations

point	S.point1()	returns the source point of S.
point	S.point2()	returns the target point of S.
bool	S.visible($point, point$)	returns true if $p2$ is visible from $p1$ with respect to P.

10.9 Algorithms for Barriers (barrier)

double S.l2_sum($facilities\&\ EX, bool\ erase, int\ choose = 1, double\ percent = 0.1, double\ accel = 1.0, double\ init\_ss = 0.01, double\ epsilon = 0.001$)

computes the optimum for problem class $1/P/(R,\infty)/l_2/\sum$ and returns the objective value.

10.10 Restriction-utilities (restr_util)

Restriction-utilities provides useful routines to handle restrictions.

double Restr_max_dir(int i, restrictions Restr)

returns the maximal coordinate in direction i ($i = 0$ x-coordinate, $i = 1$ y-coordinate).

double Restr_min_dir(int i, restrictions Restr)

returns the minimal coordinate in direction i ($i = 0$ x-coordinate, $i = 1$ y-coordinate).

double Restr_diff_value(int i, restrictions Restr)

returns the maximal difference between the coordinates in direction i ($i = 0$ x-coordinate, $i = 1$ y-coordinate).

void DrawRestr(window W, restrictions Restr)

draws the restrictions in a window W.

restrictions ReadRestr(ifstream file)

returns the restrictions given in the correct format in *file*.

restr_poly ReadPoly(ifstream file)

returns a polygon as a restriction given in the correct format in *file*.

restr_circle ReadCirc(ifstream file)

returns a circle as a restriction given in the correct format in *file*.

restr_rect ReadRect(ifstream file)

returns a rectangle as a restriction given in the correct format in *file*.

Chapter 11

Planar Classes with Gauges

11.1 Polyhedral Gauges (polygauge)

1. Definition

An instance G of the data type *polygauge* is a gauge in the two-dimensional plane defined by the sequence of its vertices in counterclockwise order which creates the cones. These cones are numbered in counterclockwise order. The number of vertices is called the size of G. See also the Gauge-utilities header (gauge_util.h), which provides Load, Save, Create and View for Gauges.

2. Creation

polygauge $G(list$<$point$>$\ pl)$;

introduces a variable G of type *polygauge*. G is initialized to the polygauge with vertex sequence pl.
Precondition: The vertices in pl are given in counterclockwise order and define a polygauge. The polyhedral belonging to the polygauge must be convex.

polygauge G;

introduces a variable G of type *polygauge*. G is initialized to the empty polygauge.

3. Operations

point	$G\ [int\ c]$	returns the c-th extreme point of G.
point	G.origin()	returns the origin G.

list<point>	G.vertices()	returns the vertex sequence of G for the reference point *origin*.
list<point>	G.vertices(*point r*)	returns the vertex sequence of G for the reference point r.
list<point>	G.vertices(*location rl*)	returns the vertex sequence of G for the reference location rl.
list<segment>	G.segments(*point r*)	returns the sequence of bounding segments of the cones of G in counterclockwise order for the reference point r.
list<segment>	G.segments(*location rl*)	returns the sequence of bounding segments of the cones of G in counterclockwise order for the reference location rl.
list<double>	G.alphas()	returns the angle sequence of G.
list<point>	G.conepoints(*point r*, *int c*)	returns the extreme points creating the cone c.
list<point>	G.conepoints(*location rl*, *int c*)	returns the extreme points creating the cone c.
list<segment>	G.coneseg(*point r*, *int c*)	returns the segments creating the cone c.
list<segment>	G.coneseg(*location rl*, *int c*)	returns the segments creating the cone c.
int	G.conetest(*point r*, *int c*, *point p*, *bool& unique*)	returns 1 if p lies inside the cone c, 0 otherwise; if p lies on a segment border, unique is false; if r is equal to p, 0 is returned and unique is false.
int	G.conetest(*location rl*, *int c*, *location l*, *bool& unique*)	returns 1 if l lies inside the cone c, 0 otherwise; if l lies on a segment border, unique is false; if rl is equal to l, 0 is returned and unique is false.
int	G.inCone(*point r*, *point p*, *bool& unique*)	returns the number of the cone, p lies inside; if p lies on a segment border, unique is false; if r is equal p, 0 is returned and unique is false.

int	G.inCone(*location rl*, *location l*, *bool& unique*)	returns the number of the cone, l lies inside; if l lies on a segment border, unique is false; if rl is equal to l, 0 is returned and unique is false.
list<point>	G.pl_inCone(*point r*, *int c*, *list<point> pl*)	returns the list of points *pl* which lie inside the cone c, returns the empty list if no point is inside the cone.
list<location>	G.ll_inCone(*location rl*, *int c*, *list<location> ll*)	returns the list of locations of *ll* which lie inside the cone c, returns the empty list if no location is inside the cone.
double	G.norm(*point r*, *point p*)	returns the norm for the gauge from point r to point p.
double	G.norm(*location rl*, *location l*)	returns the norm for the gauge from location rl to location l.
list<double>	G.lin_describe(*point r*, *int c*)	returns m (in list.head()) and b (in list.tail()) as the linear description ($y = m \cdot x + b$) of the cone c; returns inf (in list.head()) and x (in list.tail()) if segment is vertical.
list<double>	G.lin_describe(*location rl*, *int c*)	returns m (in list.head()) and b (in list.tail()) as the linear description ($y = m \cdot x + b$) of the cone c, returns inf (in list.head()) and x (in list.tail()) if segment is vertical.
double	G.max_dist()	returns the maximal euclidean distance from origin of the gauge G to the extreme points.
point	G.maxi(*point r*, *int i*)	returns the extreme point of G with the maximal coordinate, i=0 for x-coord and i=1 for y-coord.
point	G.mini(*point r*, *int i*)	returns the extreme point of G with the minimal coordinate, i=0 for x-coord and i=1 for y-coord.
point	G.maxi(*location rl*, *int i*)	returns the extreme point of G with the maximal coordinate, i=0 for x-coord and i=1 for y-coord.

point	G.mini(*location* rl, *int* i)	returns the extreme point of G with the minimal coordinate, i=0 for x-coord and i=1 for y-coord.
double	G.max_diff(*int* i)	returns the difference between $max\,i$ and min, i=0 for x-coord and i=1 for y-coord.
polygauge	G.dual()	returns the dual gauge of G.
polygauge	G.rotate(*double alpha*)	returns the gauge created by a rotation of G by angle *alpha*.
polygauge	G.l1()	returns the l_1 gauge.
polygauge	G.linf()	returns the l_∞ gauge.
polygauge	G.unit()	returns the gauge with all segments of length 1.
polygauge	G.join(*polygauge*& H)	returns the union of gauge G and H.
polygauge	G.scale(*double scale*)	returns the gauge with all segments scaled with *scale*.
polygauge	G.translate(*point* r)	returns the gauge with all segments translated to r.
bool	G.symmetrical()	returns true if G is symmetrical, false otherwise.
int	G.size()	returns the size of G.
bool	G.empty()	returns true if G is empty, false otherwise.
bool	G == H	tests for equality of G and H.
bool	G != H	tests for inequality of G and H.

11.2 Mixed Gauges (mixgauge)

1. Definition

An instance G of the data type *mixgauge* is a gauge in the two-dimensional plane defined by the sequence of its vertices in counterclockwise order which creates the cones. These cones are numbered in counterclockwise order. The number of vertices is called the size of G. See also the Gauge-utilities header (gauge_util.h), which provides Load, Save, Create and View for Gauges.

2. Creation

mixgauge G(*list<point> pl*, *list<bool> typ*);

introduces a variable G of type *mixgauge*. G is initialized to the mixgauge with vertex sequence *pl*.
Precondition: The vertices in *pl* are given in counterclockwise order and define a mixgauge. The polyhedral belonging to the mixgauge must be convex.

mixgauge G;

introduces a variable G of type *mixgauge*. G is initialized to the empty mixgauge.

3. Operations

point	G [*int c*]	returns the c-th extreme point of G .
bool	G.conetype(*int c*)	returns the type of the cone c.
list<point>	G.vertices(*point r*)	returns the vertex sequence of G for the reference point r.
list<point>	G.vertices(*location rl*)	returns the vertex sequence of G for the reference location rl.
list<segment>	G.segments(*point r*)	returns the sequence of bounding segments of the cones of G in counterclockwise order.
list<segment>	G.segments(*location rl*)	returns the sequence of bounding segments of the cones of G in counterclockwise order.
list<double>	G.alphas()	returns the angle sequence of G .
list<point>	G.conepoints(*point r*, *int c*)	returns the extreme points creating the cone c.
list<point>	G.conepoints(*location rl*, *int c*)	returns the extreme points creating the cone c.
list<segment>	G.coneseg(*point r*, *int c*)	returns the segments creating the cone c.
list<segment>	G.coneseg(*location rl*, *int c*)	returns the segments creating the cone c.

int G.conetest(*point r*, *int c*, *point p*, *bool& unique*)

returns 1 if p lies inside the cone c, 0 otherwise; if p lies on a segment border, unique is false; if r is equal to p, 0 is returned and unique is false.

int G.conetest(*location rl*, *int c*, *location l*, *bool& unique*)

returns 1 if l lies inside the cone c, 0 otherwise; if l lies on a segment border, unique is false; if rl is equal to l, 0 is returned and unique is false.

int G.inCone(*point r*, *point p*, *bool& unique*)

returns the number of the cone, p lies inside; if p lies on a segment border, unique is false; if r is equal to p, 0 is returned and unique is false.

int G.inCone(*location rl*, *location l*, *bool& unique*)

returns the number of the cone, l lies inside; if l lies on a segment border, unique is false; if rl is equal to l, 0 is returned and unique is false.

list<point> G.pl_inCone(*point r*, *int c*, *list<point> pl*)

returns the list of points which lie inside the cone c, returns the empty list if no point is inside the cone.

list<location> G.ll_inCone(*location rl*, *int c*, *list<location> ll*)

returns the list of locations which lie inside the cone c, returns the empty list if no location is inside the cone.

double G.norm(*point r*, *point p*) returns the norm for the gauge from point r to point p.

double G.norm(*location rl*, *location l*)

returns the norm for the gauge from location rl to location l.

list<double> G.lin_describe(*point r*, *int c*)

returns m (in list.head()) and b (in list.tail()) as the linear description ($y = m \cdot x + b$) of the cone c; returns inf (in list.head()) and x (in list.tail()) if segment is vertical.

list<double>	G.lin_describe(*location rl*, *int c*)	returns m (in list.head()) and b (in list.tail()) as the linear description ($y = m \cdot x + b$) of the cone c, returns inf (in list.head()) and x (in list.tail()) if segment is vertical.
point	G.maxi(*point r*, *int i*)	returns the extreme point with the maximal coordinate, i=0 for x-coord and i=1 for y-coord.
point	G.mini(*point r*, *int i*)	returns the extreme point with the minimal coordinate, i=0 for x-coord and i=1 for y-coord.
point	G.maxi(*location rl*, *int i*)	returns the extreme point with the maximal coordinate, i=0 for x-coord and i=1 for y-coord.
point	G.mini(*location rl*, *int i*)	returns the extreme point with the minimal coordinate, i=0 for x-coord and i=1 for y-coord.
double	G.max_diff(*int i*)	returns the difference between $max\, i$ and $min\, i$, i=0 for x-coord and i=1 for y-coord.
mixgauge	G.dual()	returns the dual gauge of G.
mixgauge	G.rotate(*double alpha*)	returns the gauge created by a rotation of G by angle $alpha$.
mixgauge	G.join(*mixgauge& H*)	returns the union of gauge G and H.
mixgauge	G.unit()	returns the gauge with all segments of length 1.
mixgauge	G.scale(*double scale*)	returns the gauge with all segments scaled with $scale$.
bool	G.symmetrical()	returns true if G is symmetrical, false otherwise.
int	G.size()	returns the size of G.
bool	G.empty()	returns true if G is empty, false otherwise.
bool	G == H	tests for equality of G and H.
bool	G != H	tests for inequality of G and H.

11.3 Gauge-utilities (gauge_util)

Gauge-utilities provides useful routines to handle gauges.

list<polygauge> ReadLPGauge(ifstream file)

returns the list of polygauges.

list<mixgauge> ReadLMGauge(ifstream file)

returns the list of mixgauges.

list<int> ReadGNumber(ifstream file)

returns the list of numbers to connect the locations with their gauges. Precondition: size of facilities is equal to size of numbers.

void SaveGauge(ofstream file, polygauge G)

saves the list of extremalpoints from polygauge G in *file*.

void SaveGauge(ofstream file, mixgauge M)

saves the list of extremalpoints and the conetyp from mixgauge M in *file*.

void DrawGauge(window W, polygauge G)

draws a polygauge G in a window W.

void DrawGauge(window W, mixgauge M)

draws a mixgauge M in a window W.

void ViewGauge(polygauge G)

opens a window and a panel where you can choose Load, Save or Create a polygauge.

void ViewGauge(mixgauge M)

opens a window and a panel where you can choose Load, Save or Create a mixgauge.

Input-format for the files

ReadGauge(file)	begin {polygauge}
	$x_1 \quad y_1$
	$\vdots$
	$x_n \quad y_n$
	end {polygauge}
ReadGauge(file,tl)	begin {mixgauge}
	$x_1 \quad y_1 \quad typ_1$
	$\vdots$
	$x_n \quad y_n \quad typ_n$
	end {mixgauge}
ReadLPGauge(file)	begin {polygaugelist}
	begin {polygauge}
	$x_1 \quad y_1$
	$\vdots$
	$x_n \quad y_n$
	end {polygauge}
	$\vdots$
	begin {polygauge}
	$x_1 \quad y_1$
	$\vdots$
	$x_m \quad y_m$
	end {polygauge}
	end {polygaugelist}
ReadLMGauge(file)	begin {mixgaugelist}
	begin {mixgauge}
	$x_1 \quad y_1 \quad typ_1$
	$\vdots$
	$x_n \quad y_n \quad typ_n$
	end {mixgauge}
	$\vdots$
	begin {mixgauge}
	$x_1 \quad y_1 \quad typ_1$
	$\vdots$
	$x_m \quad y_m \quad typ_m$
	end {mixgauge}
	end {mixgaugelist}
ReadGNumber(file)	begin {gaugenumber}
	num_1
	$\vdots$
	num_n
	end {gaugenumber}

Chapter 12

Graph Classes

12.1 Basic Classes for Graphs (sol_typ, edge_segment, node_weight in graphsolution.h)

1. Definition

An instance of the data type *sol_typ* consists of two integers and one double to describe the solution of non restrictive problems. *sol_typ* is used to denote a location along the edge, where the integers describe the adjacent nodes and the double is a parameter t $\in$ [0..1] to give the position along the edge.

sol_typ S;	creates an instance of *sol_typ*.
S.source:=i;	defines the startnode i of edge e.
S.target:=j;	defines the endnode j of edge e.
S.alpha_s:=t;	defines the point t on edge e.

2. Definition

An instance of the data type *edge_segment* is derived from the data type *sol_typ* with an additional double to describe the solution of restrictive problems. *edge_segment* is used to denote a part of an edge.

edge_segment ES;	creates an instance of *edge_segment*.
ES.source:=i;	defines the startnode i of edge e.
ES.target:=j;	defines the endnode j of edge e.

ES.alpha_s:=t_1; defines the starting point t_1 on edge e.

ES.alpha_t:=t_2; defines the endpoint t_2 on edge e.

3. Definition
An instance of the data type *node_weight* is a list of doubles. A *lolagraph* uses this data-type as node-type.

12.2 Lolagraph (classlolagraph)

1. Definition

An instance of the data type *lolagraph* consists of a LEDA-Graph graph<node_weight, double>.

2. Creation

lolagraph G; creates a LEDA-Graph graph<node_weight,double> G

3. Operations

3.1 Helpfunctions

matrix	G.shortest_path()	returns the distance matrix for the graph G.
edge	G.inz_edge(*node*& v, *node*& w)	returns the edge between node v and node w in graph G.

3.2 Algorithms

list<sol_typ>	G.abs_cent(*double*& *objval*)	returns the solution of the 1-center problem of a whole graph, directed and undirected.
list<sol_typ>	G.center(*double*&)	returns the solution of the 1-center problem on the nodes of a graph.
list<sol_typ>	G.median(*char wert*, *double*& *objval*)	returns the solution of the 1-median problem of the nodes for a directed or undirected graph.

list<sol_typ> G.N_median(*int* , *double&*, *list<int>& belongto*)

returns the solution of the N-median problem of the nodes for a directed or undirected graph.

list<sol_typ> G.N_partitioning(*int N*, *double& objval*, *list<int>& belongto*, *bool typ*)

returns the solution of the N-median(typ=1) or N-Center(typ=0) problem of the nodes for a directed or undirected graph, using the "node-partitioning-heuristic" .

list<sol_typ> G.Nmed_exchange(*int N*, *double& objval*, *list<int>& belongto*)

returns the solution of the N-median problem of the nodes for a directed or undirected graph, using the "exchange-heuristic" .

list<sol_typ> G.N_greedy(*int N*, *double& objval*, *list<int>& belongto*, *bool typ*)

returns the solution of the N-median (typ=1) or N-center (typ=0) problem of the nodes for an undirected graph, using the "greedy-heuristic" .

12.3 Directed Graphs (loladirected)

1. Definition

An instance of the data type *loladirected* is a directed graph derived of the data type *lolagraph*.

2. Creation

loladirected GD; creates a directed lolagraph GD.

3. Operations

list<sol_typ> GD.inmedian(*double& objval*) returns the solution of the 1-inmedian problem of the nodes for a directed graph.

list<sol_typ> GD.outmedian(*double& objval*) returns the solution of the 1-outmedian problem of the nodes for a directed graph.

12.4 Undirected Graphs (lolaundirected)

1. Definition

An instance of the data type *lolaundirected* is a undirected graph derived of the data type *lolagraph*.

2. Creation

lolaundirected GU; creates a undirected lolagraph *GU*.

3. Operations

list<sol_typ>	*GU*.abs_med(*double& objval*)	returns the solution of the 1-median problem of a whole graph.

12.5 Trees (lolatree)

1. Definition

An instance of the data type *lolatree* is a special undirected *lolagraph*, containing no loops. It is derived from the data type *lolaundirected*.

2. Creation

lolatree T; creates a lolatree *T*.

3. Operations

3.1 Helpfunctions

bool	*T*.is_tree()	tests if G is a tree, i.e. containing no loops.

3.2 Algorithms

list<sol_typ>	*T*.tree_node_med(*double& objval*)

returns the solution of the local 1-median problem on a tree.

list<sol_typ> *T*.tree_med(*double*& *objval*)

returns the solution of the absolute 1-median problem on a tree.

list<sol_typ> *T*.tree_center(*double*& *objval*)

returns the solution of the absolute 1-center problem on a tree.

list<sol_typ> *T*.two_tree_center(*double*& *objval*)

returns the solution of the absolute 2-center problem on a tree.

list<sol_typ> *T*.tree_node_center(*double*& *objval*)

returns the solution of the local 1-center problem on a tree.

12.6 Graph-utilities (graph_util)

1. Definition

An instance of the data type *graph_util* supports the data type *lolagraph*. It is derived from the data type *lolagraph*. The class *graph_util* provides input and output routines to handle the class *lolagraph*.

2. Creation

graph_util G ; creates a lolagraph as graph_util *G*.

3. Operations

Input and Output

void *G*.QGraphView(*list<vector> objvec*, *string normact*, *facilities*& *EX*, *list<string>*& *Loctxt*, *list<edge_segment>*& *EL*, *bool dir*)

draws a lolagraph.

lolagraph *G*.CreateGraph(*string locfile*, *string adjfile*, *facilities*& *EX*) creates a lolagraph file and returns a parameterized graph and the facilities EX.

void	G.SaveGraph($ofstream\&\, file$, $facilities\&\, EX$, $list$<$string$>$\&\, Loctxt$)	
		saves a lolagraph in a file.
void	G.SaveAdjlist($ofstream\&\ \ file$)	
		saves the adjacent list of a lolagraph in a file.
graph_util&	G.ReadGraph($ifstream\&\, file$, $facilities\&\, EX$, $list$<$string$>$\&\, Loctxt$)	
		reads a lolagraph file and returns a parameterized graph and the facilities EX.

Chapter 13

Discrete Classes

13.1 Discrete Locations (discrete)

1. Definition

An instance of the data type *discrete* two list of location giving the positions for the demand and the supply points, and a matrix with travelling costs from a supply point to a demand point.

2. Creation

discrete $D(matrix\ Cost)$;

introduces a variable D of type *discrete*. D is initialized with the cost matrix M.

3. Operations

3.1 Input and Output

void	D.ReadDiscrete($ifstream\&\ file$)	reads all data for D from $file$.
void	D.DiscView($double\ objval$)	shows the results and locations of the current problem.
void	D.SaveDisc($ofstream\&\ file$)	saves all data of D into $file$.

3.2 Algorithms

double D.UFLP_greedy()

returns the solution of the UFL problem using the "greedy-heuristic" .

double D.UFLP_stingy()

returns the solution of the UFL problem using the "stingy-heuristic" .

double D.UFLP_Interchange()

returns the solution of the UFL problem using the "interchange-heuristic" .

double D.UFLP_dualoc()

returns the solution of the UFL problem using the dualoc as an exact algorithm .

Chapter 14

User Interface Class to LOLA

14.1 Algorithm (planealg)

1. Definition

An instance of the data type *planealg* consists of the algorithm to solve planar location problems.

2. Creation

planealg P;	creates a variable P of type *planealg*.

3. Operations

list<*location*>	P.alg_solution()	returns the solution for the current problem.
void	P.WriteOpt(*ostream*& *out*)	writes the solution for the current problem.

3.1 Algorithms

3.1.1 Median-problems

double	P.l1_sum(*facilities*& *EX*)	computes the optimum for problem class $1/P/./l_1/\sum$ and returns the objective value.
double	P.l1_sum(*facilities*& *EX*, *restrictions*& *R*)	computes the optimum for problem class $1/P/R = convex/l_1/\sum$ *(Konstukrionslinien-algorithmus)* and returns the objective value.

double P.l2sqr_sum($facilities\&\ EX$)

computes the optimum for problem class $1/P/./l_2^2/\sum$ and returns the objective value.

double P.l2sqr_sum($facilities\&\ EX,\ restrictions\&\ R$)

computes the optimum for problem class $1/P/R = convex/l_2^2/\sum$ and returns the objective value.

double P.linf_sum($facilities\&\ EX$) computes the optimum for problem class $1/P/./l_\infty/\sum$ and returns the objective value.

double P.linf_sum($facilities\&\ EX,\ restrictions\&\ R$)

computes the optimum for problem class $1/P/R = convex/l_\infty/\sum$ and returns the objective value.

double P.l2_sum($facilities\&\ EX,\ double\ epsilon$)

computes the optimum for problem class $1/P/./l_2/\sum$ (Weiszfeld-Algorithm) and returns the objective value.

double P.l2_sum($facilities\&\ EX,\ barrier\&\ R$)

computes the optimum for problem class $1/P/(R,\infty)/l_2/\sum$ and returns the objective value, where (R,∞) may be **one** barrier such as a circle, a segment or a polygon.

double P.l2_sum($facilities\&\ EX,\ barrier\&\ R, bool\ erase, int\ choose, double\ percent,$ $double\ accel, double\ init\_ss, double\ epsilon$)

computes the optimum for problem class $1/P/(R,\infty)/l_2/\sum$ and returns the objective value, where (R,∞) may be **one** barrier such as a circle, a segment or a polygon. Same algorithm like the above one but user can choose from two heuristics(0 or 1) and define a percentage(percent) of points to handle, acceleration factor(accel) and inital step size(init_ss) as well as epsilon for the iterations.

double P.lp_sum($facilities\&\ EX,\ int\ p, double\ epsilon, double\ delta, int\ iter\_max$)

computes the optimum for problem class $1/P/./l_p/\sum$ (Generalized Weiszfeld-Algorithm) and returns the objective value.

double P.lp_sum($facilities$& EX, $restrictions$& R, $int\ p, double\ epsilon,$ $double\ delta, int\ iter\_max$)

computes the optimum for problem class $1/P/R/l_p/\sum$ and returns the objective value.

double P.l2sqr_qsum($facilities$& EX)

computes the optimum for problem class $1/P/./l_2^2/Q-\sum$ (Q-criterial problem) and returns the objective value.

double P.l1_2sum($facilities$& EX)

computes the optimum for problem class $1/P/./l_1/2-\sum$ (bi-criterial problem) and returns the objective value.

double P.linf_2sum($facilities$& EX)

computes the optimum for problem class $1/P/./linf/2-\sum$ (bi-criterial problem) and returns the objective value.

double P.N_l1_sum($facilities$& EX, $int\ n$, $matrix$& w)

computes the optimum for problem class $N/P/./l_1/\sum$ and returns the objective value.

double P.N_linf_sum($facilities$& EX, $int\ n$, $matrix$& w)

computes the optimum for problem class $N/P/./l_\infty/\sum$ and returns the objective value.

double P.N_l2sqr_sum($facilities$& EX, $int\ n$, $matrix$& w)

computes the optimum for problem class $N/P/./l_2^2/\sum$ and returns the objective value.

double P.N_l2sqr_sum($facilities$& EX, $int\ n$, $matrix$& w)

computes the optimum for problem class $N/P/./l_p/\sum$ (Version 1) and returns the objective value.

double P.N_l2sqr_sum($facilities$& EX, $int\ n$, $matrix$& w)

computes the optimum for problem class $N/P/./l_p/\sum$ (Version 2) and returns the objective value.

double P.L_l2_sum_M3($facilities$ &EX)

computes the optimum for problem class $1L/P/./l_2/\sum$ and returns the objective value.

double P.L_l1_sum_M3($facilities$ &EX)

computes the optimum for problem class $1L/P/./l_1/\sum$ and returns the objective value.

double P.L_linf_sum_M3($facilities$ &EX)

computes the optimum for problem class $1L/P/./l_\infty/\sum$ and returns the objective value.

double P.L_lp_sum_M3($facilities$ &EX, *int p*)

computes the optimum for problem class $1L/P/./l_p/\sum$ and returns the objective value.

double P.L_l2_sum_M2logM($facilities$ &EX)

computes the optimum for problem class $1L/P/./l_2/\sum$ with $O(M^2 logM)$ and returns the objective value.

double P.L_l2_sum_M2($facilities$ &EX)

computes the optimum for problem class $1L/P/v_i = 1/l_2/\sum$ with $O(M^2)$ and returns the objective value.

double P.L_RkonvPoly_l2_sum(*facilities*& EX, *restrictions*& R)

computes the optimum for problem class $1L/P/R = convex/l_2/\sum$ and returns the objective value.

double P.L_RkonvPoly_lp_sum(*facilities*& EX, *restrictions*& R, *int p*)

computes the optimum for problem class $1L/P/R = convex/l_p/\sum$ and returns the objective value.

double P.L_RkonvPoly_l1_sum(*facilities*& EX, *restrictions*& R)

computes the optimum for problem class $1L/P/R = convex/l_1/\sum$ and returns the objective value.

double P.L_RkonvPoly_linf_sum($facilities$& EX, $restrictions$& R)

computes the optimum for problem class $1L/P/R = convex/l_{inf}/\sum$ and returns the objective value.

double P.L_RkonvPoly_block_sum($facilities$& EX, $restrictions$& R, $Blocknorm$ &B)

computes the optimum for problem class $1L/P/R = convex/l_{block}/\sum$ and returns the objective value.

3.1.2 Center-problems

double P.l1_max($facilities$& EX)

computes the optimum for problem class $1/P/./l_1/max$ and returns the objective value.

double P.l1_max($facilities$& EX, $restrictions$& R)

computes the optimum for problem class $1/P/R = convex/l_1/max$ and returns the objective value.

double P.l1_v1_max($facilities$& EX)

computes the optimum for problem class $1/P/v_i = 1/l_1/max$ and returns the objective value.

double P.l2_max($facilities$& EX)

computes the optimum for problem class $1/P/./l_2/max$ and returns the objective value.

double P.l2_max($facilities$& EX, $restrictions$& R)

computes the optimum for problem class $1/P/R = \text{convex polyhedron}, v_i = 1/l_2/max$ and returns the objective value.

double P.linf_max($facilities$& EX)

computes the optimum for problem class $1/P/./l_\infty/max$ and returns the objective value.

double P.linf_max($facilities$& EX, $restrictions$& R)

computes the optimum for problem class $1/P/R = convex/l_\infty/max$ and returns the objective value.

double P.linf_v1_max($facilities$& EX)

computes the optimum for problem class $1/P/v_i = 1/l_\infty/max$ and returns the objective value.

double P.elzhearn($facilities$& EX)

computes the optimum for Elzinga-Hearn-Algorithm and returns the objective value.

double P.N_linf_max($facilities$& EX, *int* N, *matrix*& W)

computes the optimum for problem class $N/P/./l_\infty/max$ and returns the objective value.

double P.N_l1_max($facilities$& EX, *int* N, *matrix*& W)

computes the optimum for problem class $N/P/./l_1/max$ and returns the objective value.

double P.L_l2_max_M4($facilities$ &EX)

computes the optimum for problem class $1L/P/./l_2/max$ and returns the objective value.

double P.L_l1_max_M4($facilities$ &EX)

computes the optimum for problem class $1L/P/./l_1/max$ and returns the objective value.

double P.L_linf_max_M4($facilities$ &EX)

computes the optimum for problem class $N/P/./l_\infty/max$ and returns the objective value.

double P.L_l2_max_MlogM($facilities$ &EX)

computes the optimum for problem class $1L/P/v_i = 1/l_2/max$ with $O(MlogM)$ and returns the objective value.

double P.L_l2_max_M2logM($facilities$ &EX)

computes the optimum for problem class $1L/P/./l_2/max$ with $O(M^2logM)$ and returns the objective value.

double P.L_lp_max_M4(*facilities* &EX, *int* p)

computes the optimum for problem class $1L/P/./l_p/max$ and returns the objective value.

14.2 Algorithm (lgraphalg)

1. Definition

An instance of the data type *lgraphalg* consists of the algorithms to solve location problems with a network or graph.

2. Creation

lgraphalg GrA; creates a variable GrA of type *lgraphalg*.

3. Operations

list<sol_typ> GrA.alg_solution() returns the solution for the current problem.

void GrA.WriteOpt(*ostream*& *out*)

writes the solution for the current problem.

3.1 Algorithms

double GrA.inmedian(*loladirected*& G)

computes the In-Median for problem class $1/G_D/./d(V,V)/\sum$ and returns the objective value.

double GrA.outmedian(*loladirected*& G)

computes the Out-Median for problem class $1/G_D/./d(V,V)/\sum$ and returns the objective value.

double GrA.median(*loladirected*& G)

computes the median for problem class $1/G_D/./d(V,V)/\sum$ and returns the objective value.

double GrA.median(*lolaundirected*& U)

computes the optimum for problem class $1/G/./d(V,V)/\sum$ and returns the objective value.

double GrA.abs_inmedian(*loladirected*& G)

computes the In-Median for problem class $1/G_D/./d(V,G)/\sum$ and returns the objective value.

double GrA.abs_outmedian(*loladirected*& G)

computes the Out-Median for problem class $1/G_D/./d(V,G)/\sum$ and returns the objective value.

double GrA.abs_median(*loladirected*& G)

computes the median for problem class $1/G_D/./d(V,G)/\sum$ and returns the objective value.

double GrA.abs_median(*lolaundirected*& U)

computes the optimum for problem class $1/G/./d(V,G)/\sum$ and returns the objective value.

double GrA.center(*lolaundirected*& G)

computes the optimum for problem class $1/G/./d(V,V)/\max$ and returns the objective value.

double GrA.center(*loladirected*& G)

computes the optimum for problem class $1/G_D/./d(V,V)/\max$ and returns the objective value.

double GrA.abs_center(*lolaundirected*& U)

computes the optimum for problem class $1/G/./d(V,G)/\max$ and returns the objective value.

double GrA.abs_center(*loladirected*& G)

computes the optimum for problem class $1/G_D/./d(V,G)/\max$ and returns the objective value.

double GrA.loc_tree_median($lolatree$& U)

computes the optimum for problem class $1/T/./d(V,V)/\sum$ and returns the objective value.

double GrA.abs_tree_median($lolatree$& U)

computes the optimum for problem class $1/T/./d(V,T)/\sum$ and returns the objective value.

double GrA.loc_tree_center($lolatree$& U)

computes the optimum for problem class $1/T/./d(V,V)/\max$ and returns the objective value.

double GrA.abs_tree_center($lolatree$& U)

computes the optimum for problem class $1/T/./d(V,T)/\max$ and returns the objective value.

double GrA.tree_two_center($lolatree$& U)

computes the optimum for problem class $2/T/./d(V,T)/\max$ and returns the objective value.

double GrA.N_median_cplex($lolaundirected$& $U, int\ n, list$<int>& $belongto$)

computes the optimum for problem class $N/G/./d(V,V)/\sum$ using cplex and returns the objective value.

double GrA.N_median_partitioning($lolaundirected$& $U, int\ n, list$<int>& $belongto$)

computes the optimum for problem class $N/G/./d(V,V)/\sum$ using the node-partitioning-heuristic and returns the objective value, in the list 'belongto' the membership-information for every node is contained.

double GrA.N_median_exchange($lolaundirected$& $U, int\ n, list$<int>& $belongto$)

computes the optimum for problem class $N/G/./d(V,V)/\sum$ using the exchange-heuristic and returns the objective value, in the list 'belongto' the membership-information for every node is contained.

double *GrA*.N_median_greedy(*lolaundirected& U, int n, list<int>& belongto*)

computes the optimum for problem class $N/G/./d(V,V)/\sum$ using the greedy-heuristic and returns the objective value, in the list 'belongto' the membership-information for every node is contained.

double *GrA*.N_center_partitioning(*lolaundirected& U, int n, list<int>& belongto*)

computes the optimum for problem class $N/G/./d(V,V)/\max$ using the node-partitioning-heuristic and returns the objective value, in the list 'belongto' the membership-information for every node is contained.

double *GrA*.N_center_greedy(*lolaundirected& U, int n, list<int>& belongto*)

computes the optimum for problem class $N/G/./d(V,V)/\max$ using the greedy-heuristic and returns the objective value, in the list 'belongto' the membership-information for every node is contained.

list<vector> *GrA*.medPareto_bi(*lolaundirected& U, list<edge_segment>& wholesol*)

computes the optimum for problem class $1/G/./d(V,G)/2-\sum_{par}$.

list<vector> *GrA*.medPareto_bi(*lolaundirected& U, list<edge_segment>& wholesol*)

computes the optimum for problem class $1/G/./d(V,G)/Q-\sum_{par}$.

list<vector> *GrA*.medPareto_bi(*lolaundirected& U, list<edge_segment>& wholesol*)

computes the optimum for problem class $1/G_D/./d(V,G)/Q-\sum_{par}$.

list<vector> *GrA*.medPareto_bi(*lolaundirected& U, list<edge_segment>& wholesol*)

computes the optimum for problem class $1/G_D/./d(V,V)/Q-\sum_{par}$.

list<vector> *GrA*.medPareto_bi(*lolaundirected& U, list<edge_segment>& wholesol*)

computes the optimum for problem class $1/G/./d(V,V)/Q-\sum_{par}$.

list<vector> *GrA*.medPareto_bi(*lolaundirected*& *U*, *list<edge_segment>*& *wholesol*)

computes the optimum for problem class $1/G_D/./d(V,V)/Q - \max_{par}$.

list<vector> *GrA*.medPareto_bi(*lolaundirected*& *U*, *list<edge_segment>*& *wholesol*)

computes the optimum for problem class $1/G/./d(V,V)/Q - \max_{par}$.

list<vector> *GrA*.medPareto_bi(*lolaundirected*& *U*, *list<edge_segment>*& *wholesol*)

computes the optimum for problem class $1/G_D/./d(V,G)/Q - \max_{par}$.

list<vector> *GrA*.medPareto_bi(*lolaundirected*& *U*, *list<edge_segment>*& *wholesol*)

computes the optimum for problem class $1/G/./d(V,G)/Q - \sum_{lex}$.

list<vector> *GrA*.medPareto_bi(*lolaundirected*& *U*, *list<edge_segment>*& *wholesol*)

computes the optimum for problem class $1/G_D/./d(V,G)/Q - \sum_{lex}$.

list<vector> *GrA*.medPareto_bi(*lolaundirected*& *U*, *list<edge_segment>*& *wholesol*)

computes the optimum for problem class $1/G_D/./d(V,V)/Q - \sum_{lex}$.

list<vector> *GrA*.medPareto_bi(*lolaundirected*& *U*, *list<edge_segment>*& *wholesol*)

computes the optimum for problem class $1/G/./d(V,V)/Q - \sum_{lex}$.

list<vector> *GrA*.medPareto_bi(*lolaundirected*& *U*, *list<edge_segment>*& *wholesol*)

computes the optimum for problem class $1/G_D/./d(V,V)/Q - \max_{lex}$.

list<vector> *GrA*.medPareto_bi(*lolaundirected*& *U*, *list<edge_segment>*& *wholesol*)

computes the optimum for problem class $1/G/./d(V,V)/Q - \max_{lex}$.

14.3 Algorithm (gaugealg)

1. Definition

An instance of the data type *gaugealg* consists of the algorithm to solve planar location problems with gauges as the distance functions.

2. Creation

gaugealg GA; creates a variable GA of type *gaugealg*.

3. Operations

list<location> GA.alg_solution() returns the solution for the current problem.

void GA.WriteOpt(*ostream& out*)

writes the solution for the current problem.

Algorithms

double GA.sum(*list<polygauge>& LG*, *list<int>& Ln*, *facilities& EX*)

computes the optimum for problem class $1/P/./\gamma/\sum$ and returns the objective value.

double GA.bi_crit_sum(*list<polygauge>& LG*, *list<int>& Ln*, *facilities& EX*)

computes the optimum for problem class $1/P/./\gamma/2-\sum_{par}$ and returns the number of different situations of the solution.

double GA.L_block_sum_M3(*list<polygauge> &LG*, *list<int> &Ln*, *facilities &Ex*)

$1L/P/./\gamma_B/\sum$ with $O(M^3)$.

double GA.L_block_max_M4(*list<polygauge> &LG*, *list<int> &Ln*, *facilities &Ex*)

$1L/P/./\gamma_B/\max$ with$O(M^4)$.

double GA.L_RkonvPoly_block_sum(*list<polygauge> &LG*, *list<int> &Ln*, *facilities EX*, *restrictions &Restr*)

$1L/P/./\gamma_B/\max$ with $O(M^4)$.

Chapter 15

LOLA Error Messages

The LOLA specific error messages are numbers of 4 digits (XYZZ). Each digit has a special meaning.

X identifies the general problem class in which the function that was called is included.

Y identifies the class in which the function that was called is included.

ZZ identifies the function in which the error occurred.

Here we give a table of all currently used error numbers.

X	Y	ZZ	error in
1	Y	ZZ	LOLA planar library
2	Y	ZZ	LOLA graph library
3	Y	ZZ	LOLA discrete library
1	0	ZZ	LOLA frontend
1	1	ZZ	class loc_vector
1	2	ZZ	class location
1	3	ZZ	class facilities
1	4	ZZ	restrictions
1	5	ZZ	class facs_util
1	6	ZZ	gauge_util
1	7	ZZ	gauges
1	8	ZZ	barriers
2	0	ZZ	class lolagraph
2	1	ZZ	class graph_util
2	2	ZZ	class lolatree
3	1	ZZ	discrete

In the following each error code will be specified in detail.

X	Y	ZZ	error in
1	0	01	_algoch ()
1	0	02-12	main ()
1	1	01	locvector :: rotate ()
1	2	01	location :: norm ()
		02	location :: r_angle ()
		03	location :: loc2point ()
		04	location :: loc2vector ()
		05	location :: test_coords ()
		06	location :: test_weights ()
		07,08	location :: operator[]
		09,10	location :: operator()
		11	location :: operator/
		12	location :: WriteAsPolygonList
1	3	01,02	facilities :: n_median_objective ()
		03	facilities :: median_objective ()
		04	facilities :: WriteOpt ()
		05	facilities :: remove ()
		06	facilities :: N_lp_sum ()
		07	facilities :: linf_v1_max ()
		08-30	facilities :: N_linf_max_mps ()
		31-33	facilities :: N_l1_sum ()
		34,35	facilities :: refname ()
		36-37	facilities :: N_linf_max ()
		38-41	facilities :: gauge_sum ()
		42-46	facilities :: gauge_bi_crit_sum ()
1	4	01	restr_circle :: restr_circle ()
		02-04	check_simplicity ()
		05	restr_poly :: proj_v1_l2_max ()
		06	retsr_rect :: proj_v1_l2_max ()
		07	restr_rect :: operator[]
		08	restr_rect :: operator()
		09,10	ReadRestr ()
		11	restriction :: proj_v1_l2_max ()
		12	inside_all ()
		13	polygon_barrier :: polygon_barrier ()
		14	polygon_barrier :: polygon_barrier
1	5	01-07	fac_util :: ReadLoc ()
		08-11	fac_util :: ReadMat ()
		12	fac_util :: SaveLoc ()
		13	fac_util :: SaveLoc_Res ()
		14	fac_util :: View ()
1	6	01,02	ReadGNumber ()
		03-06	ReadGauge ()
		07,08	ReadLPGauge ()
		09,10	ReadLMGauge ()

		11	ViewGauge ()
1	7	01-03	polygauge :: polygauge ()
		04	polygauge :: scale
		05	polygauge :: operator []
		06,07	polygauge :: conepoints
		08,09	polygauge :: coneseg ()
		10,11	polygauge :: conetest ()
		12	polygauge :: pl_inCone
		13	polygauge :: ll_inCone
		14	polygauge :: maxi
		15	polygauge :: mini
		16	polygauge :: max_diff
		17-21	mixgauge :: mixgauge ()
		22	mixgauge :: operator [] ()
		23	mixgauge :: conetype ()
		24,25	mixgauge :: conepoints ()
		26,27	mixgauge :: coneseg ()
		28,29	mixgauge :: conetest ()
		30	mixgauge :: pl_inCone ()
		31	mixgauge :: ll_inCone ()
		32	mixgauge :: maxi ()
		33	mixgauge :: mini ()
		34	mixgauge :: max_diff ()
		35	mixgauge :: scale ()
1	8	01,02	barrier :: l2_sum ()
		03	halfspace :: compute ()
		04	tree :: flow_rank ()
		05	tree :: crash_rank ()
		06	sbo :: opt_shdw_segments ()
2	0	01	lolagraph :: shortest_path ()
		02	lolagraph :: inz_edge ()
		03	lolagraph :: parlocation ()
2	1	01	graph_util :: LGraphView ()
		02	graph_util :: ReadAdjlist
		03	graph_util :: SaveAdjlist ()
		04	graph_util :: SaveGraph ()
		05-12	graph_util :: ReadGraph ()
2	2	01	lolatree :: lolatree ()
		02-04	lolatree :: tree_med ()
3	1	01	discrete :: UFLP_dualoc ()
		02	discrete :: SaveDisc ()
		03-05	discrete :: disc_read ()
		06-07	discrete :: Read_discrete ()

IMPRESSUM

Coordinators:

Horst W. Hamacher, Holger Hennes, Kathrin Klamroth, Martin C. Müller, Stefan Nickel and Anita Schöbel.

Contributors:

Ingo Bartling, Sonja Becker, Annette Brockmann, Chokri Hamdaoui, Holger Hennes, Karin Jung, Jörg Kalcsics, Kathrin Klamroth, Ralf Leipe, Martin C. Müller, Stefan Nickel, Michael A. Ochs, Lothar Schnell, Karen Schulze, Yingxin Wang and Ansgar Weißler.

Home-page: http:\\www.mathematik.uni-kl.de\~lola

E-mail: lola@mathematik.uni-kl.de

Tel: (+49)/631/205-4558, (+49)/631/205-2511

Fax: (+49)/631/29081