Routing with Genetic Algorithms*

– A Toolbox –

Christian Matuszewski[1]

Frank Schmiedle[2]

Robby Schönfeld[1]

July 1999

Technical Report

Advisors: Prof. Dr. Bernd Becker[2]
Nicole Drechsler[2]
Dr. Rolf Drechsler[2]
Prof. Dr. Paul Molitor[1]

[1] Universität Halle-Wittenberg
Institut für Informatik
Lehrstuhl für Technische Informatik

[2] Universität Freiburg
Institut für Informatik
Lehrstuhl für Rechnerarchitektur

* Authors and advisors appear in alphabetical order.

Contents

1 Introduction

The present report[1] gives an overview and a description of a software environment for *Routing with Genetic Algorithms*.

Within the scope of a research project supported by the DFG[2] under grant *Be1176/12-1* and *Mo645/5-1* this work was carried out as a joint project of the research groups of Prof. Dr. Bernd Becker at the Albert-Ludwigs-University of Freiburg and Prof. Dr. Paul Molitor at the Martin Luther University of Halle-Wittenberg from 1998 to 1999.

The authors describe the present status of the software they have developed, including internal data structures, main features and some basic interfaces. They also provide a user manual in order to simplify handling. All tools have been implemented in a C/C++ environment, if not explicitly stated otherwise.

In Chapter 2 the routing environment `MaDRE` is presented. Within the same data structure several routing strategies are included as *Basic Optimization Modules* (BOMs). BOMs are essential for the heuristic learning approach (Chapter 4). Furthermore, an implementation of a completely different routing strategy, the *Exact Routing* approach, is introduced.

Genetic Algorithms (GAs) are the key issue of Chapter 3. The software library `GAME` that provides an environment to perform genetic algorithms is described. It allows a wide range for choosing parameters, genetic operators, selection/survival strategies etc. Also a mapping on a parallel computer system is possible.

`MaDRE` and `GAME` as introduced in Chapters 2 and 3 are combined for the heuristic learning approach. Chapter 4 describes an implementation that has been produced so far.

In Chapter 5 we provide a collection of routing instances that are to be used for benchmarking our methods. Finally we present a tool for visualization that enables the user to illustrate the routed channels generated by the diverse strategies.

[1] The TR is available as TR 125 from Institute of Computer Science at University of Freiburg

[2] **D**eutsche **F**orschungs**g**emeinschaft

2 MaDRE — Multiple Strategy Detailed Routing Environment

The routing environment `MaDRE` that is used in the project is a derivative of `CHANNLIB` [1]. Several new features have been added.

This chapter presents `MaDRE` in detail. In Section 2.1, an efficient data structure for representing instances of the *Channel Routing Problem* (CRP), a special file format for interchanging routing instances, and data structures used for representing channel properties as, e.g., lower bounds of the channel width, are presented. Then, we focus on some implementation details with respect to the initialization of the data structures (Section 2.2). In Section 2.3 we present the routing algorithms implemented in the routing environment. In particular, we give criteria determining the time to break off a routing strategy in order to let another routing strategy doing the remaining job. These criteria have to diagnose whether the running routing strategy does work efficiently on the specific routing instance. The chapter closes with the description of a completely different routing approach. The goal of the *Exact Routing* approach is the search for all solutions to a given CRP instance. Here, symbolic methods are used.

Further details and background information can be found in [2, 3, 4, 5, 6, 7, 8].

2.1 Data structures, formats and elementary concepts

2.1.1 Basic data structures

In general, an instance of CRP consists of the representation of the routing area and the netlists (pin locations, net affiliation). Furthermore, implicitly a certain routing model is assumed.

In `MaDRE`, the routing may only consist of vertical and horizontal wire segments connected by vias, different routes are required to be edge-disjoint, and two routes may cross over at a grid point, but they may not bend away from each other, i.e., knock-knees are explicitly forbidden. With respect to these restrictions we underlayed a grid based model for the basic data structure of the routing area. This grid is stored as 3-dimensional matrix (track, column, layer). At each grid point some information has to be stored. Let us give the most important features in the following. For illustration we refer to Figure 2.1.

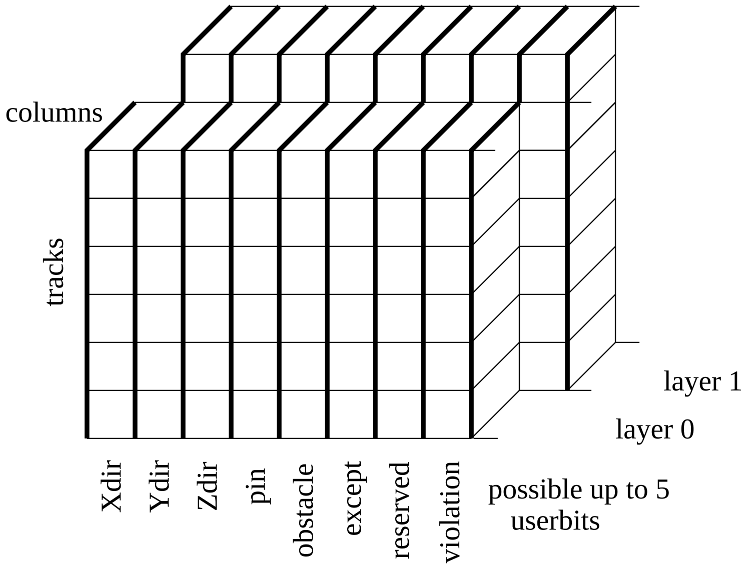


Figure 2.1: Data structure of the routing area.

- A grid point has to cover the situations that the grid point is not occupied or that a wire, a pin or an obstacle is located at it. Thus, at each grid point, we have reserved five bits `Xdir`, `Ydir`, `Zdir`, `pin`, and `obstacle`. The first three bits give information whether the grid point is a wire or not. A non zero bit denotes the fact that the present grid point is connected to the direct neighbor in positive X,Y or Z-direction, respectively.

- As we allow special routers (see, e.g., the random router presented in Section 2.3.8) and genetic algorithms to generate illegal wirings of routing instances during evolution, our data structure has to handle multi-occupied grid points, i.e., grid points temporarily shared by different nets. A `violation`-bit at each grid point indicates whether this grid point is such a *multipoint*. The names of the nets sharing such a grid point and the corresponding directions of the wires are stored in a hash table using the coordinates of the grid points as keys.

- At each grid point five `userbits` are available which can be freely used by the routers. Furthermore, a `reserved`-bit and `except`-bit for internal use is part of the grid point data structure.

Additionally, in order to speed up the access on the data – note that the grid can be quite large –, the netlists are stored in three dictionaries: the `net`-, the `pin`-, and the `point`-dictionary. The `net`-dictionary uses the netlist number for storing the number of nets contained in the corresponding netlist. To access information on a specific net, a key consisting of the netlist number and the number of the specific net is used. This information comprises the netlist number, the number of the net, the name of the net, and the number of pins of the net. The `pin`-dictionary stores the pin locations of the pins which are addressed by the netlist number, the net number, and the pin number. The `point`-dictionary allows to efficiently access information on a specific grid point.

We call the representation of a (prerouted) channel a `TASK`. A `TASK` contains as data

- `CHANNEL task.channel`
  the matrix for representing the routing area;
- `PINLIST pinlist`
  the identifier of the task. `PINLIST` is of type `long int`;
- `BOOLEAN defined`
  the validity of the task;
- `fitness quality`
  the quality of the wiring (see Section 2.1.3);
- `char NAME[20]`
  the basename of the loaded CRP;
- `VisitCard VC`
  the visiting card of the task which stores the properties of the CRP.

In order to facilitate the work with dictionaries, `MaDRE` makes available different global variables and functions, namely

- `int size`
  the size of a dictionary is a global variable which is limited to 4001;
- `void InitChannNets ()`
  initializes the `net-`, the `pin-` and the `point`-dictionary;
- `void FinishChannNets ()`
  frees allocated memory occupied by the dictionaries;
- `void InitChannPin ()`
  initializes an additional dictionary for internal use;
- `void FinishChannPin ()`
  frees allocated memory of the additional dictionary;
- `int NumberOfNets (PINLIST pinlist)`
  returns the number of nets of the task specified by `pinlist`;
- `void InsertPin (PINLIST pinlist, int netname, POINT point)`
  stores the position `point` as a pin from net `netname` in the dictionaries;

- `BOOLEAN NetNumberToNet`
      `(PINLIST pinlist, int netnumber, NET *net)`
  stores a pointer to the net record of the net with internal number `netnumber` in `net` and returns `TRUE`. If an error occurs, value `FALSE` is returned and `net` is set to value `NoNet`;

- `BOOLEAN NameToNet`
      `(PINLIST pinlist, int netname, NET *net)`
  stores a pointer to the net record of the net with name `netname` in `net` and returns `TRUE`. If an error occurs, value `FALSE` is returned and `net` is set to value `NoNet`;

- `void NetToInfo (NET net, NetInfo *netinfo)`
  copies the information about a net located at position `net` to `*netinfo`;

- `netinfo.netname`
  the name of the net (see also the `PIN`-keyword in Section 2.1.2);

- `netinfo.netnumber`
  the internal number of the net;

- `netinfo.npins`
  the number of pins that belong to the net;

- `BOOLEAN NetToPoint (NET net, int pinnr, POINT *point)`
  stores the coordinates of a pin at location `*point` and returns `TRUE`, if `net` is an existing net and `pinnr` is a number of a pin of this net;

- `BOOLEAN PointToNet`
      `(PINLIST pinlist, POINT point, NET *net, int *pinnr)`
  if a pin exists at the coordinates of `point`, `net` and `pinnr` are set to the corresponding values and value `TRUE` is returned.

Representation of violations

As already mentoined before, violations, i.e., multipoints, cannot be represented in the channel matrix in the same way like points that are occupied by only one single net. The reason for this is that in the channel data for matrix points, for each direction there is only one flag that indicates whether there is a wire segment towards that direction or not. It is not possible to store information about how much different wire segments of the same direction there are neither which is the net these segments belong to.

For this, data structures called `MP` and `MultiPoint` are defined in `MaDRE`:

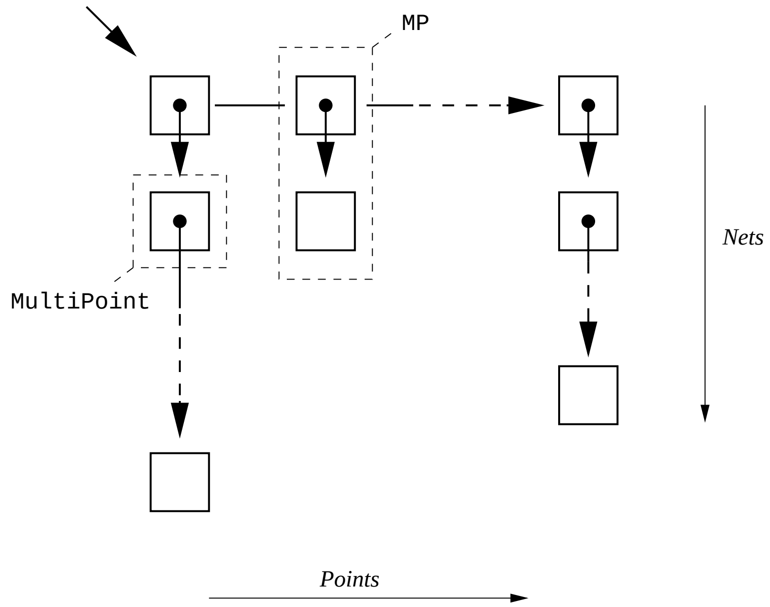


Figure 2.2: internal representation of multipoints

```
typedef struct MultiPoint {
  int    netname;
  BITSET direction;
  struct MultiPoint *next;
} *MP;
```

`MP` is a pointer to a list of lists of `MultiPoint` items. Each of the `MultiPoint` item lists contains all the wires located at a certain point p in the channel matrix. For each net that occupies p there is one item including the number of the net, the directions of all wires for this net at p and a pointer to the next item in the list. Figure 2.2 illustrates this.

The `violation` bit is set for p to mark it as a multipoint and as an effect of this the bits `Xdir`, `Ydir`, `Zdir` for p are considered as invalid. The information which position in the `MP` list corresponds to which net is stored in the appropriate dictionaries.

Furthermore,, `MaDRE` makes available some functions facilitating the handling of multipoints:

- `void InitMultiNet()`
  initializes dictionaries for storing the information on multipoints;
- `void FinishMultiNet(PINLIST plist)`
  frees the allocated memory of the multipoint dictionaries;
- `void InsertMultiWire(PINLIST pinlist, POINT p, int netname,`

`BITSET dir)`
inserts a multi-occupancy for grid point `p` of net `netname` with directions `dir`;

- `void DeleteMultiWire(PINLIST pinlist, POINT p, int netname)`
  deletes a occupancy of net `netname` for grid point `p`;

- `BOOLEAN PointToDirection(PINLIST pinlist, POINT p, int netname,BITSET *dir)`
  if `p` is multi-occupied from net `netname`, `dir` is set to the corresponding directions and value `TRUE` is returned;

- `long int NumberOfMultiPoints()`
  returns the total number of current multipoints.

2.1.2 The CIO file format

The interchange of data between routing tools and the users is done via a special file format, the *Channel-Input-Output* (CIO) file format. A CIO–file is organized as follows:

`CHANNEL`	$channelname$
`MAX`	$x_{max}\ y_{max}\ z_{max}$
`PIN`	$net_{name}\ \ x\ y\ z$
	.
	.
	.
`OBSTACLE`	$x_1\ y_1\ z_1\ \ x_2\ y_2\ z_2$
	.
	.
	.
`WIRE`	$x_1\ y_1\ z_1\ \ x_2\ y_2\ z_2$
	.
	.
	.
`CONTACT`	$x\ y\ z$
	.
	.
	.
`MULTIPOINT`	$net_{name}\ \ x\ y\ z\ \ dir_{biteset}$
	.
	.
	.
`CHANNEL`	

The keyword `CHANNEL` denotes the beginning and the end of the specification. Memory for the matrix will be allocated for x_{max} columns, y_{max} tracks and z_{max} layers. `MAX` describes the upper right corner of the routing area and its maximum value is set to $(8192, 4096, 16)$. A pin is specified by the name of its net net_{name} and its coordinates (x, y, z) in the routing area. `OBSTACLES` are forbidden areas for routing. They are specified by their lower left (x_1, y_1, z_1) and upper right (x_2, y_2, z_2) corner. The wiring consists of straight horizontal and vertical lines (`WIRE`) and vias (`CONTACT`), which connect adjacent grid points of different layers. The keyword `MULTIPOINT` enables multi-occupancy of grid point (x, y, z) from net net_{name} with wires encoded in the `BITSET` $dir_{biteset}$.

Two functions for loading and saving a CRP, respectively, are made available by `MaDRE`:

- `BOOLEAN ReadChannel(TASK *task, char *filename)`:
  loads a CRP from the file `filename` and initializes the basic data structure for the CRP, the `task`. If any error occurs, then value `FALSE` is returned, otherwise value `TRUE`.

- `BOOLEAN WriteChannel(TASK *task, char *filename)`:
  saves the `task` in the file `filename`. If any error occurs, then value `FALSE` is returned, otherwise value `TRUE`.

2.1.3 Computation of channel properties

Routing models

Due to the fact that parallel overlapping wire segments in adjacent layers interact, the technologies applied nowadays alternatively use layers on which only horizontal or vertical wire segments, respectively, are allowed. That is, *H-layers*, which only contains horizontal wire segments, take turns with *V-layers* which only contain vertical wire segments.

In contrast to this, different routing models are allowed in `MaDRE`:

- `FREE`
  no additional restrictions are made. Horizontal and vertical wire segments may share the same layers.

- `CHANNELCONVENTION`
  the top and the bottom track must not contain horizontal wire segments.

- `MANHATTAN`
  alternating layer assignment with one of the following submodels:

  - `HV` H-layers have even and V-layers odd numbers.

- `VH` V-layers have even and H-layers odd numbers.
- `AUTO` overlap model is determined by prerouting. If no prerouting is present the model is chosen such that the layer with the most terminals is a vertical layer. This strategy saves vias.

Visiting card

To each task a `VisitCard` for classification of the routing instance is associated by `MaDRE`. This data structure collects information about the routing area, the design rules, the net classes, lower bounds on the channel width, and the *Vertical Constraint Graph* (VCG) and its properties.

After loading a CRP, the visiting card should be initialized and the corresponding data should be computed. This can be done by setting task.defined to value `FALSE` and then calling the function `FillupVC`.

- `BOOLEAN FillupVC (TASK *task)` computes the properties of the CRP and put the data in the `VisitCard`.

The various entries in the visiting card can be divided into the following five subtopics:

routing area
Here, we store several features of the routing area, like locations of pins and obstacles in general and the type of prerouting as described in Section 2.1.3.

design rules
Here, we note information on the maximum length of overlapping wire segments between (adjacent and nonadjacent) layers, whether stacked vias are allowed, and so on.

net classes
This data area contains information on the number of nets, the sizes of the nets, i.e., the numbers of pins, and so on. By splitting the multiterminal nets into two-terminal nets, the nets can be classified by the location of their pins as shown at an example in Figure 2.3.

lower bounds
Here, lower bounds for the CRP are stored. We use *density*, *macle*[1] and *flux*. The (closed) density of a CRP is the maximum number of nontrivial nets splitted by any vertical cut of the channel. The macle is the longest path in the VCG. The flux is the minimum number of tracks which is needed to perform all column changes to the right or left.

[1] **Ma**ximum **C**hain **Le**ngth

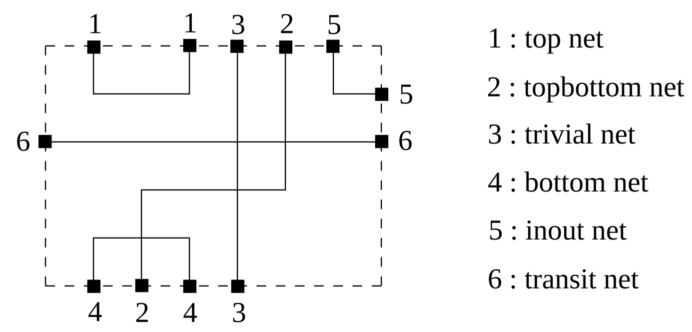


Figure 2.3: Net classes.

VCG
The VCG is a special representation of relations between nets used by most channel routers. Let i and j be two nets with $i \neq j$, a directed edge (i, j) exists, if a terminal of net i is located at the upper channel border in a column in which a terminal of net j is located at the lower channel border. Additionally, we store some properties of the VCG, e.g., whether the VCG is trivial (contains no edges) or whether it is cyclic.

Fitness

The quality of the wiring is assembled in the structure `fitness`. We compute the total length of wires, the number of vias of each single net and the whole task, respectively. Additionally, we compute the minimal bounding box of routing area used for the wiring. The essential functions for initialization, calculation, and displaying the fitness of a CRP are the following:

- `void init_fitness (TASK *task, fitness *fit)`
  initializes the fitness structure for `task` and stores the location in `*fit`. This function must be called for every new task with `init_fitness (&task, &(task.quality))`.
- `void compute_fitness (TASK *task)`
  computes the quality of the wiring of the CRP `task`. The result is stored in the fitness record `task.quality`.
- `void show_fitness (fitness fit)`
  displays the content of the fitness record. The results of each single net and the overall wiring are shown.
- `void show_short_fitness (fitness fit)`
  displays only the results for the overall wiring.

- `void clear_fitness (TASK *task, fitness *fit)`
  reset the entries of the fitness record `fit`.
- `void copy_fitness (fitness *dest, fitness *source)`
  copies the content of the fitness record from `source` to `dest`.

2.2 The function main()

Because of the data structures used, some initialization procedures have to be called at the beginning of the program. To form an idea, we refer to the following code sequence which is an example for a program which applies a router of `MaDRE`. In this code, `Router()` is a dummy and must be replaced by an actual routing function described in the Sections 2.3.1– 2.3.8.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>
#include "misc.h"
#include "longset.h"
#include "diction.h"
#include "ngraphs.h"
#include "channpoi.h"
#include "channnet.h"
#include "channel.h"
#include "channpin.h"
#include "taskdef.h"
#include "channasc.h"
#include "channsca.h"
#include "fitness.h"

int main ()
{
    TASK    task;
    char    *in_file  =  "in.cio",
            *out_file = "out.cio";

    size = 224;
    InitChannNets ();
    InitChannPin ();

    if (ReadChannel(&task, in_file))
      {
            task.defined = FALSE;
            FillupVC (&task);
```

```
            init_fitness (&task, &(task.quality));

            ROUTER();

            WriteChannel (&task, out_file);
            FinishTask (&task);
      }
    FinishChannNets ();
    FinishChannPin ();
}
```

The function

- `void FinishTask (TASK *task)`
  frees the allocated memory of the specified `VisitCard`, the fitness records, and the matrix. Before calling `FinishTask`, the functions `ReadChannel`, `FillupVC` and `init_fitness` must have been called.

2.2.1 User manual

Now we outline general command line parameters of our routing environment `MaDRE` and its effects. The syntax of the `MaDRE`–command is as follows:

```
MaDRE [--help]|[options] source [-odest]
```

The option `--help` sends a short description to `stdout`. For all other options, a CIO-file `source` is expected by the `MaDRE` program. The output file `dest` is optional. The router to be applied, its parameters, and the general parameters are given by the setting of the `options`. At first let us have a look at the parameters for setting up the routing and the overlap model.

`-laFREE`: routing model is `FREE`.

`-laHV`: routing model is `MANHATTAN` and overlap model is `HV`.

`-laVH`: routing model is `MANHATTAN` and overlap model is `VH`.

`-laAUTO`: routing model is `MANHATTAN`. The overlap model is determined by existing prerouting.

`-laNOCHANNEL`: reset the flag for channel convention `CHANNCONV`.

Options for displaying channel and wiring properties follow now:

`-r`: displays the fitness for each single net and of overall wiring.

`-rs`: only displays the fitness of overall wiring.

`-sv`: displays the contents of `VisitCard`.

`-co`: checks routing correctness and displays possible errors and warnings. Occurring errors are incomplete wiring, short circuits, blind components, and illegal pins. Warnings are caused by nets with only one pin, dead ends and wire cycles.

Some additional flags for displaying the routing progress and for sorting netlists are made available:

`-v`: verbose mode displays information of routing progress.

`-d`: debug mode displays routing in detail.

`-s`: sorts nets and pins by pin coordinates in ascending order.

2.3 Routing strategies implemented as BOMs

In this section we present several routing strategies integrated in `MaDRE`. Further details on the methods introduced in Sections 2.3.1 – 2.3.7 can be found in [2, 3].

2.3.1 Constraint leftedge algorithm

The *leftedge edge* algorithm was introduced by Hashimoto and Stevens [9] to route channels without vertical constraints but can easily be adapted to do so.

We will describe the algorithm shortly to be able to explain the parameters and the truncation criteria of the router. In the first step of the algorithm, the horizontal net segments are sorted by the x-coordinate of their leftmost terminal. Then, beginning with the topmost track and the leftmost net segment all tracks are filled such that no segments overlap and no vertical constraint is violated.

We extended the algorithm by using the *routing sequence* parameter which was first introduced by Deutsch [10] for the dogleg router. This parameter specifies from which channel side (top or bottom) and in which order the nets are assigned to tracks. There is no need to start in the top-left corner as the original algorithm does. One can start in every other corner and further, one can change the side and net order during the routing. Deutsch used only the eight following routing sequences which always alternate the sides: top-left↔bottom-left, top-left↔bottom-right, top-right↔bottom-right, top-right↔bottom-left, bottom-left↔top-left, bottom-left↔top-right, bottom-right↔top-right, bottom-right↔top-left. Alternating the sides often avoids an agglomeration of net-segments on one side.

To have more freedom in learning of parameters we added the non-alternating routing sequences top-left, top-right, bottom-left, bottom-right, and the routing sequences that alternate only the net orders but not the channel side: top-left↔top-right, top-right↔top-left, bottom-left↔bottom-right, bottom-right↔bottom-left. Additionally we added the special sequence “best”. If this sequence is specified, the channel is routed with all routing sequences and the best result is chosen and returned.

To combine several routers we want to measure the quality produced by the router during its work and break it off if poor results are produced. The following three criteria are suitable for the leftedge router (see [2] for details).

track usage
For a compact routing it is necessary to use the given area as much as possible. After the routing of a track is complete, we count the columns in which the track is used by a wire. If this value is below a given threshold (parameter `min_used_track`), the routing of the track is discarded and the router stops.

density
The (global) density of a channel is defined as the maximum of the local densities in every column. Since the density is a lower bound of the width of the channel we can estimate the minimum width after the routing of a track. If this estimated width is larger than the maximum of the lower bounds density and macle, we discard the currently routed track and stop the routing. To use this truncation criterion, set the parameter `test_for_density_violation` to `TRUE`.

macle
Since the macle is a lower bound of the channel width, we can use it as a truncation condition. After the routing of a track, the new macle indicates if it is possible to route the channel in $\max\{old\_macle, density\}$ tracks or not. If not, we discard the track and break off the execution of the router. The parameter `est_for_macle_violation` has to be set to `TRUE` if you want to use this feature.

```
void LeftEdgeConstr (TASK    *Task,
                     int      overlap_model,
                     routing_sequence rs,
                     int      min_used_track,
                     BOOLEAN test_for_density_violation,
                     BOOLEAN test_for_macle_violation);
```

The command line to call the constrained leftedge algorithm is given by

```
MaDRE [-lc] [-seSEQUENCE]
```

Description of parameters:

`-lc`: calls the constraint leftedge algorithm.

`-seSEQUENCE`: sets routing sequence. Possible values are `TL`, `BL`, `TR`, `BR`, `TL_BL`, `TL_BR`, `TR_BL`, `TR_BR`, `BL_TL`, `BL_TR`, `BR_TL`, `BR_TR`. `TL_BR`, e.g., is an alternating mode. It starts to route the top track from left, continues with the bottom track from right and so on. The default value is `TL`.

2.3.2 Dogleg router

The *dogleg* router from Deutsch [10] is very similar to the leftedge router. The only difference is that all nets are splitted into two-point nets such that the horizontal net segment of one net has not to be routed entirely in one track. This can reduce the length of the longest path in the VCG and therefore better routings are possible.

A parameter to control the behavior of the dogleg router is the *range* parameter. This parameter specifies the minimum number of consecutive net segments of one net which have to lie entirely in one track. Exceptions of this rule occur if the net has not enough two-point nets (trivial) and if a net is partially routed and the rest of the net has not enough two-point nets. So, a large value of the range parameter produces routings with few contacts, while a smaller value can reduce the number of used tracks. If the range parameter is set to the special value 0, the router is forced to route the entire net in one track. Then, the dogleg router behaves like the constraint leftedge router. There are two range parameters in the function call of the dogleg router. Value `range` has to be less than or equal to `range2`. If both values differ then the router is executed with every value from `range` to `range2` and the best routing result is returned.

The routing sequence parameter is the same as for the leftedge algorithm. We made the observation that with two of the additional routing sequences we can route the difficult example by Deutsch in 20 tracks [2]. Till now it was common knowledge that the dogleg router needs 21 tracks [11].

The truncation criteria are the same as for the leftedge router.

```
void dogleg (TASK   *task,
             int     overlap_model,
             routing_sequence rs,
             int     range,
             int     range2,
             int     min_used_track,
             BOOLEAN test_for_density_violation,
             BOOLEAN test_for_macle_violation);
```

The command line to call the dogleg router is given by

```
MaDRE [-do] [-raINTEGER] [-reINTEGER]
```

Description of parameters:

`-do`: calls the dogleg router.

`-raINTEGER`: start value of the range.

`-reINTEGER`: end value of the range.

2.3.3 Yoshimura's router

The router of Yoshimura [12] also fills one track at once. The difference to the previous channel routers is that the nets that will be routed in the track are determined by a weighting function. The weight W(n) will be assigned to every net and is defined by:

$$W(n) = \sum_{z \in Z(n)} U(z) + (P \cdot L(n) + D(n)) \cdot |Z(n)|$$

Thereby $Z(n)$ is the set of zones which contain net n. Note, that the notion of the zone [13, 14] is fundamental for this and the next channel router. A zone is basically a large-as-possible contiguous set of columns, in which the horizontal net segments mutually overlap. Thus, if we assign the nets to tracks, we do not have to consider every column. It is enough to assure that all nets of one zone lie in different tracks. Furthermore, the equation $U(z) = 2 \cdot D_z$ holds where D_z is the local density of zone z. $L(n)$ is the length of the longest path in the VCG which contains n and $D(n)$ is the outdegree of n. The parameter P controls the influence of the length of the longest path compared to the other values. Yoshimura suggests a value of 3 for P. As with the range parameter of the dogleg router, if the values for `P1` and `P2` are different, the best solution is returned.

Since the order of the horizontal net segments plays no role in the assignment to the tracks, there are only four appropriate routing sequences: top, bottom, top↔bottom, bottom↔top.

Furthermore, we can tell the router to use doglegs (`dogleg == TRUE`), no doglegs (`nondoglegs == TRUE`), or to use doglegs but reduce them in a previous step (`reduce_doglegs == TRUE`). The reduction of doglegs works in the following way. If the merging of two consecutive subnets of one net would not enlarge the longest path in the VCG through one of both nodes then these nets are merged. This procedure reduces the number of doglegs and the running time. If more than one of the strategies is selected, we return the best solution found.

The truncation criteria are the same as the criteria of the leftedge and dogleg routers.

```
void YoshimuraRouter (TASK   *Task,
                      int     overlap_model,
                      BOOLEAN nondogleg,
                      BOOLEAN dogleg,
                      BOOLEAN reduce_doglegs,
                      routing_sequence rs,
                      int     P1,
                      int     P2,
                      int     min_used_track,
                      BOOLEAN test_for_density_violation,
                      BOOLEAN test_for_macle_violation);
```

The command line to call Yoshimura's router is given by

```
MaDRE [-y]|[-yd]|[-yr]|[-yndr] [-seSEQUENCE] [-raINTEGER]
                    [-reINTEGER]
```

Description of parameters:

`-y`: calls Yoshimuras router without doglegs.

`-yd`: calls Yoshimuras router with doglegs.

`-yr`: calls Yoshimuras router with reduced doglegs.

`-yndr`: finds best of.

`-seSEQUENCE`: sets routing sequence. Possible values are `T, B, B_T, T_B, BEST`.

`-raINTEGER`: start value for P.

`-reINTEGER`: end value for P.

2.3.4 Net-merge-router of Yoshimura and Kuh

The router of Yoshimura and Kuh [14] is very different from the previous channel routers. The routers just described work on the channel track by track, while the router of Yoshimura and Kuh tries to simplify the VCG by merging the nodes and then assigns a track to every node.

The merging is done in the following way. All zones in the channel are scanned from left to right, starting with the leftmost zone. Let L be the set of all nets which end in the currently treated zone or end left from this zone. Let R be the set that contains all nets which start in the zone right to the current zone. Now we try to merge nets in L with nets in R, i.e., we try to find nets which will be routed together in one track. Thereby the nets are chosen by a heuristics such that the new macle of the VCG is not (much) enlarged.

During the processing of the zones we do not have to start in zone one. We obtain better results when starting in a zone with high local density and proceeding to the right and then proceeding from the rightmost zone to zone one. Thus, the router has the two parameters `start_column1` and `start_column2`, and if `typ == zone` then both parameters are interpreted as zones. If `typ == column` then the `start_column` values are interpreted as columns and will be converted into zones. If both values differ, then the router will be executed with all zones (columns) between `start_column1` and `start_column2` and the best routing result will be returned.

As with Yoshimura's router, we can specify whether we want to route with doglegs, reduced doglegs, or without doglegs.

The truncation criteria of the Net-Merge-Router are different from the criteria of the other channel routers because we do not proceed the channel track by track.

density

If two nets are merged, a (small) gap is possibly left between them. This area is lost for the router. We can use this information to give a lower bound of the channel width after routing. If this lower bound exceeds the lower bound we get before starting the router, the execution of the routing is stopped when parameter `test_for_density_violation` is set to `TRUE`.

macle

If the parameter `test_for_macle_violation == TRUE` then after each net-merging the new macle of the longest path in the VCG is compared to the maximum of the lower bounds of the channel width. If the new macle is larger than the lower bound, the routing is stopped.

After breaking off the routing, we look for the path in the VCG which contains the most merged nets. We route all nets in this path except the unmerged ones.

```
void YoshimuraKuh (TASK      *Task,
                   int        algorithm,
                   int        overlap_model,
                   BOOLEAN    nondogleg,
                   BOOLEAN    dogleg,
                   BOOLEAN    reduced_dogleg,
                   start_typ  typ,
                   int        start_column1,
                   int        start_column2,
                   int        edges_per_node,
                   BOOLEAN    test_for_density_violation,
                   BOOLEAN    test_for_macle_violation);
```

The command line to call the router of Yoshimura and Kuh is given by

```
MaDRE [-yk]|[-ykd]|[-ykr] [-raINTEGER] [-reINTEGER] [-stc]|[-stz]
```

Description of parameters:

`-yk`: calls router of Yoshimura and Kuh without doglegs.

`-yd`: calls router of Yoshimura and Kuh with doglegs.

`-ykr`: calls router of Yoshimura and Kuh with reduced doglegs.

`-raINTEGER`: start value for start zone or start column.

`-reINTEGER:` end value for end zone or end column.

`-stc:` column mode. The router interpretes values for `ra` and `re` as columns. This is the default mode.

`-stz:` zone mode. The router interpretes values for `ra` and `re` as zones.

The parameters specifying the truncation criteria of the constrained leftedge algorithm, the dogleg router, the router of Yoshimura, and the router of Yoshimura and Kuh are given by

```
MaDRE [-uINTEGER] [-td] [-tm]
```

Description of parameters:

`-uINTEGER:` lower bound for occupied columns.

`-td:` switches on `test_density_violation`.

`-tm:` switches on `test_macle_violation`.

2.3.5 Lineprobe router

We have implemented a lineprobe router based on the ideas presented in [15] and [16]. This algorithm has been generalized to multilayer routing. This technique tries to find a connection between two grid points in the routing area by repeated line spreading. The `source` and the `target` points are the origins of the first horizontal and vertical line segments, which are stored in separate lists `slist` and `tlist`. We choose one or more *escape points* from each line for generating new lines perpendicular to it. In the following we have a closer look at the choice of escape points.

Escape strategy

The choice of the escape points for generating new lines is controlled by several strategies which have a strong influence on the time and space complexity of the lineprobe algorithm. We present four strategies for chosing the escape points:

everygrid
Every grid point on the line is an escape point. The runtime is very high. However, a solution is found in the case that a solution exists.

longestline
The point which allows to generate the longest new line is the escape point. This reduces mainly the occupied memory space.

endpoint
The endpoints of the lines are taken as the escape points. This strategy results in the fastest execution time of the lineprobe algorithm but also generates the worst wiring results.

hitobstacle
If the line hits an obstacle, we use strategy endpoint otherwise longestline. Thereby, we hope that the wiring is clinging to the obstacles and routing area is saved.

The command line parameter of the `MaDRE` environment for the escape strategies can be set as follow:

```
MaDRE [-lpl]|[-lpg]|[-lpe]|[-lpo]
```

Description of parameters:

`-lpg`: lineprobe router with escape strategy **everygrid**

`-lpl`: lineprobe router with escape strategy **longestline**

`-lpe`: lineprobe router with escape strategy **endpoint**

`-lpo`: lineprobe router with escape strategy **hitobstacle**

Net scheduling – Net ordering

The lineprobe router (and also the maze routers described in Sections 2.3.7, 2.3.6) are very sensitive to the net ordering. That is why we offer the following net scheduling strategies.

1. in sequence of the input
2. random order
3. sort by size of bounding box
4. sort by number of pins
5. sort by size of bounding box and number of pins

The strategies 3, 4, and 5 are available in ascending and descending order. A special net scheduler function is utilized for all routers to minimize the implementation effort:

- `void Scheduler(PINLIST pinlist, N_Strategy net_strategy, Scheduler_RouteOneNet net_router, BOOLEAN IsRestricted)`
  The scheduler function expects the identifier of the task `pinlist`, one of the net strategies, and a pointer to a function `net_router`. The `IsRestricted` flag is set for routing into `MANHATTAN` model.

The `RouteOneNet` function is the implementation of a router, which is able to handle the entire net. The main task of this function is to divide multiterminal nets into two point nets by using the requested net strategy and to execute a function `RouteOneConnection` for each two point net.

The command line parameter of the `MaDRE` environment for the several net strategies can be set as follow:

```
MaDRE [-ns]|[-nr]|[-nbu]|[-nbd]|[-npu]|[-npd]|[-nou]|[-nod]
```

Description of parameters:

-ns: sequential net order

-nr: random net order

-nbu: size of bounding box upward

-nbd: size of bounding box downward

-npu: number of pins upward

-npd: number of pins downward

-nou: bounding box and pins upward

-nod: bounding box and pins downward

Pin scheduling – Splitting multiterminal nets

The basic algorithms of the maze routers and the lineprobe router can only connect two point nets, i.e., multiterminal nets have to be divided by any heuristics before routing. We have implemented four pin scheduling strategies, namely:

1. in sequence of the input,
2. focused routing I, with possible unreachable focus point,
3. focused routing II,
4. minimal spanning tree.

Methods 2 and 3 are computing the center of gravity of a net (focus). All pins of the net are tried to be routed to this focus. If the focus is unreachable, because it is isolated by obstacles, method 2 will fail. A better approach is method 3 which returns the nearest reachable point with respect to the focus as the new focus. Method 4 is the minimal spanning tree heuristics presented in [17].

Next, we describe the command line parameters of `MaDRE` for pin strategies:

```
MaDRE [-ps]|[-pf1]|[-pf2]|[-pm]
```

Description of parameters:

`-ps`: sequential pin order

`-pf1`: focused routing I

`-pf2`: focused routing II

`-pm`: minimal spanning tree

Finally, we give the function header of the lineprobe router which shows necessary parameters for calling the router within the main function as explained by the example in Section 2.2.

- ```
  void lineprobe_router(TASK *task,
       int routing_model, BOOLEAN channel_connvention,
       int overlap_model,
       P_Strategy pin_strategy, N_Strategy net_strategy,
       int normaltax, int layertax, int columntax,
       L_Strategy line_strategy,
       BreakCriterium *break_crit)
  ```

  `*task` is the pointer to the CRP to be routed. `routing_model`, `overlap_model`, and the flag `channel_convention` can be set to the values described in Section 2.1.3. The cost vector (`normaltax`, `layertax`, `columntax`) is not used in the lineprobe router. The pointer `*break_crit` refers to a data structure of the truncation criteria.

Truncation criteria

An essential property of our routing environment for learning heuristics is the possible truncation of execution of each single router by several criteria. In this chapter we present the truncation criteria used by the lineprobe router and the maze routers.
  ```

relative detour
the actual length of wires with respect to the Manhattan distance. We compute the relative detour for each single `connection`, for each single `net`, and the whole `task`.

failed connections
number of failed connections per `net` and with respect to the whole `task` in relation to the total number of connections.

failed nets
number of failed nets in relation to the number of all nets.

In the `MaDRE` environment we initialize the truncation criteria to default values so that all criteria are neutralized. The following command line parameters allows to set the truncation criteria to user-specific values:

```
MaDRE [-bdcDOUBLE] [-bdnDOUBLE] [-bdtDOUBLE] [-bcnDOUBLE]
      [-bctDOUBLE] [-bnDOUBLE] [-bb]
```

Description of parameters:

`-bdcDOUBLE`: relative detour per connection

`-bdnDOUBLE`: relative detour per net

`-bdtDOUBLE`: total relative detour
`DOUBLE` is a value between 1.0 and 100.0. Value 1.0 means no detour is allowed, 100.0 disables the criterion.

`-bcnDOUBLE`: failed connections ratio per net

`-bctDOUBLE`: total failed connections ratio

`-bnDOUBLE`: failed nets ratio
`DOUBLE` is a value between 0.0 and 1.0. Value 0.0 signifies that no connection may fail, Value 1.0 disables the criterion.

`-bb`: truncates the routing, if any criterion applies. When the option is not used the router skips to the next net in this case.

2.3.6 Lee maze router

The maze router was first presented in [18] for planar routing only. We have implemented a 3D maze router with alterable wave depth, which is based on this algorithm. The wave expansion starts at the source point and adds all unoccupied adjacent points – the *wavepoints* – to the *wavefront*. The next wavefront contains all unoccupied adjacent points of the previous wavepoints. All wavepoints are labeled with the number of the current spreading cycle as visited. If the wave depth, which is identical to the number of the current cycle, reaches its preset maximum count we choose the cheapest point from the wavefront in respect to the target and restart the wave expansion at it. Due to this a *cost function* is needed. We simply use the weighted *Manhattan distance*. If the wavefront hits the target, we trace back to the source using the labels of the wavepoints. Note, that the algorithm can possibly fail. If no member of the wavefront has an unoccupied neighbor the new wavefront is empty. This mostly happens for small values of wave depth.

Command line parameters for setting up the cost vector and the maximal wave depth:

```
MaDRE [-MINTEGER][-cINTEGER][-tINTEGER][-lINTEGER]
```

Description of parameters:

`-MINTEGER`: calls the 3D maze router with the preset wave depth `INTEGER`.

`-cINTEGER`: the weight for the costs of a column change.

`-tINTEGER`: the weight for the costs of a track change.

`-lINTEGER`: the weight for the costs of a layer change.
`INTEGER` is a value between 0 and ∞.

The cost vector parameters are optional, `MaDRE` sets the default value $(1, 1, 20)$. All known pin and net scheduling strategies are available for the maze routers too.

Finally the function header of the Lee maze router for calling within the main function looks as follows:

- ```
  void sym_maze_router(TASK *task,
       int routing_model, BOOLEAN channel_connvention,
       int overlap_model,
       P_Strategy pin_strategy, N_Strategy net_strategy,
       int normaltax, int layertax, int columntax,
       int wavedepth,
       BreakCriterium *break_crit)
  ```

  `*task` is the pointer to the CRP, which should be routed. `routing_model`, `overlap_model`, and the flag `channel_convention` can be set to the values described in Section 2.1.3. `normaltax` sets the costs of a track change. `layertax` and `columntax` are the additional costs of layer and column changes. The pointer `*break_crit` refers to a data structure of truncation criteria (see chapter 2.3.5).
  ```

2.3.7 Korn maze router

The Korn maze algorithm [19] tries to compensate the high space and run time requirement of the ordinary maze routing, which is due to the large number of visited grid points. The wavefront is initialized with source and the cheapest point with respect to the target is selected from it in each iteration. This point is deleted from the wavefront and its unoccupied adjacent grid points are added to the wavefront. If all neighbors are occupied the next cheapest point from the wavefront is selected. This is done until the wavefront reaches the target or it is empty. The visited points are labeled with gradients, which refer back to its predecessor. These labels are needed for tracing back. Note, that always a solution is found if one exists because the wavefront contains visited untraced points until all grid points of the routing area have been examined.

The command line parameters for calling the maze router and setting up the cost vector looks as follows:

```
MaDRE [-m][-cINTEGER][-tINTEGER][-lINTEGER]
```

Description of parameters:

`-m`: calls the depth first search maze router.

`-tINTEGER`: the weight for the Manhattan distance.

`-cINTEGER`: the additional tax for a column change.

`-lINTEGER`: the additional tax for a layer change.
`INTEGER` is a value between 0 and `MAX_INT`.

Finally, the function header of the Korn maze router for calling within the main function must have the following format:

- ```
  void maze_router(TASK *task,
        int routing_model, BOOLEAN channel_convention,
        int overlap_model,
        P_Strategy pin_strategy, N_Strategy net_strategy,
        int normaltax, int layertax, int columntax,
        BreakCriterium *break_crit)
  ```

  `*task` is the pointer to the CRP, which should be routed. `routing_model`, `overlap_model`, and the flag `channel_convention` can be set to the values described in chapter 2.1.3. `normaltax` sets the weight for the Manhattan distance, `layertax`, and `columntax` are the additional taxes for layer and column changes. The pointer `*break_crit` refers to a data structure of truncation criteria (see chapter 2.3.5).
  ```

2.3.8 Random router

The basic idea of the *Random Router* approach is completely different from the strategies that are fundamental for the routers previously described. This idea is summarized as follows: The nets are considered in sequential order. For routing a net, the random router first selects two pins and tries to connect them. Afterwards, all the pins consecutively are selected and for each of them the nearest point of the already existing wiring for this net is computed. The router then tries to connect the selected pin with this computed point.

A single connection between two points[2] is created by starting at the location of the selected pins followed by subsequent insertions of wire segments between the point where the wire actually ends and one of its neighbour points. The way in which the router decides which of the neighbours to choose is fundamental. So-called fitness values are assigned to the neighbours. These values depend on whether the target point is closer to the neighbour than to the actual point or not, whether the neighbour is located on a different layer[3] and also whether the neighbour point is already occupied by some other nets. Fitness values for neighbour cells are directly proportional to the probability for selection as next connecting point.

A preliminary version of the Random Router was introduced in [20, 4] where it has been used to generate routing solutions that served as starting-points for a GA. Meanwhile, we have extended it by several useful features and adapted it in a way that makes it possible to include this router into the `MaDRE` environment.

Features

We have spent considerable effort to include as many of the features of the routers in Section 2.3.1–2.3.7 as possible in order to be able to compare the results of the different routers. Thus, we implemented our random router `RND_ROUT` in a way that all of the routing models and overlap models, respectlively, that have been introduced in Section 2.1.3, can be handled. The appropriate model may be selected via command line parameters (see the listing of the parameters at the end of this section).

Further on, the user can choose an arbitrary initialization value for the random number generator in order to get some alternative solutions for one single channel with the same strategy.

Additionally, `RND_ROUT` contains a feature that is not included in any of the other routers described in this chapter: in some cases, the router allows so-called *violations*, i.e. multiple wire segments belonging to different nets in the same cell of the channel matrix[4]. These violations may be produced while a connection is being established.

[2] I.e. two pins or a pin to a point containing a wire segment of the considered net.
[3] In this case, a via has to be inserted into the channel.
[4] Such cells are called *multipoints* (see Section 2.1.1)

Of course, a channel which contains violations is no legal solution to the CRP. However, these solutions can be very useful since sometimes legal solutions of high quality can be obtained by performing few correction steps after routing with the random router.

Finally, various truncation criteria may be applied while routing a channel. RND_ROUT includes all the criteria that can be used for the lineprobe router and that are explained in Section 2.3.5. In addition to this, it is possible to define limits on the number of multipoints per connection, per net or per channel. Routing is terminated if one of these bounds is exceeded.

Data structures

Two C++ classes and several global functions have been developed and together they form our random router implementation.

In the following, we will briefly describe the purpose of the classes RANDROUT and ROULETTE and also their fundamental class member variables and member functions.

We start with the class RANDROUT. It includes all the essential information for routing and the member functions perform the appropriate channel manipulations. We begin with listing the primary class member variables:

- `TASK *task`:
  includes most of the information about the channel like pin locations, existing wires, multipoints, obstacles etc. The data structure TASK is described in Section 2.1.1

- `int fitVal[4][8]`:
  contains the possible fitness values for the neighbours of certain points. These fitness values have already been mentioned at the beginning of this section. To read the appropriate fitness value from this array, the first index value must denote the actual global routing mode while the second one has to be set according to some properties of the considered neighbour point.

- `PINREC *allPins`:
  a list of all pins that there are in the channel. This record contains the coordinates of a pin and and additionally three flags that indicate if the pin is located on the top of the channel, if it has already been considered for routing and finally if connecting it to the net it belongs to failed.

The other class member variables include parameter settings that have been made by the user or store intermediate results that are essential to decide whether some truncation criteria hold or not.

Description of member functions:

- `void select_term(NET net, int pins_on_top, int pins_on_bottom, POINT& term1, POINT& term2) const`:
  is called whenever a completely unrouted net is considered. For generation of the first connection for such a net, a selection of two pins belonging to this net is required. These pins are selected here and returned via `term1` and `term2`.

- `POINT further_term(NET net, int unrouted_pins) const`:
  is used if already some connections for a given net exist and further ones have to be established. Then only *one* pin has to be selected. This function performs the required selection and returns the selected pin.

- `POINT next_point(NET net,const POINT term) const`:
  finds among the points in the channel matrix containing a wire segment of `net` the one that is closest to `term` and returns this point. By this means, the destination of a connection between a pin and an already existing part of a net is computed.

- `int connect(int mode, const NET net, const POINT source,const POINT target)`:
  tries to connect `source` and `target` with wire segments of `net`. The given `mode` influences the fitness values that are assigned to the neighbour points. The return value contains some status information.

- `int remove_branch(NET net, POINT source)`:
  deletes the wire segments of `net` beginning at `source`. This is necessary if connecting `source` to some destination point has failed. Again, status information is returned.

- `int rand_rout()`:
  rules the interaction of the above described functions such that routing for a given channel is performed.

The class `ROULETTE` can be used for the selection of one item out of several candidates according to some fitness value that is assigned to each of them. `RND_ROUT` uses an object of this class during the generation of connections when a direction for the next wire segment has to be selected. It includes two class members:

- `int size`:
  the number of items that are candidates for selection

- `int *fitness`:
  a list of the fitness values for the candidates

There is only one member function that should be mentioned:

- `int roulette() const`:
  this function determines a “winner” and returns its index.

Besides the functions of the classes described above, there are some global functions that belong to `RND_ROUT`, too. Most of them concern multipoint handling, e.g. there are additional insertion and deletion functions for wires at multipoints. By using these functions, insertion and deletion of multiwires becomes much more convenient.

User manual

`RND_ROUT` can be called within the `MaDRE` system, but it is also possible to run it stand-alone. In the latter case, the command line must have the following format:

```
rnd_rout [--help] | [<inputfile>] [-o<outputfile>] [-rr<randominit>] [-m]
         [-co] [-laHV] [-laVH] [-laAUTO] [-laFREE] [-laMANHATTAN]
         [-laNOCHANNEL] [-laCHANNCONV] [-r] [-rs] [-v]
         [-b{d{c<value>,n<value>,t<value>},m{c<value>,n<value>,
          t<value>}, c{n<value>,t<value>},n<value>]
         [-p0,1,2,3,4,5,6,7,8,9,A,B<value>]
```

Description of parameters:

`-rr<randominit>`: initializes the random number generator.

`-m`: enables use of multipoints for routing.

`-p<index><value>`: sets fitness values for different kinds of neighbour points.

All the other parameters have the same meaning as the corresponding parameters in `MaDRE` and are described in the Sections 2.2.1 and 2.3.5, respectively.

2.4 Exact routing

All routing strategies included in MaDRE until now exclusively use heuristics for the routing process trying to generate one possible solution. We also implemented a program EXACT that is able to do *Exact Channel Routing*. Currently EXACT is not fully integrated in the MaDRE environment. Nevertheless, basic interfaces have been chosen consistent with MaDRE, so that an integration (in the near future) should be feasible.

Carrying out exact routing means that for a given channel, a set of *all* possible solutions to the CRP is generated. This is a difficult problem and thus computation is possible for small problem instances only. It is done by using *symbolic representation* for the channel, i.e. MDD[5] variables represent cells of the channel matrix. The theoretical part of this method is not considered here since it is described in [5]. Here, we focus on some implementation issues of Version 2.3. of EXACT.

At first we will describe the data structures used for the symbolic channel representation and also those used for the routing process. Interfaces of some basic functions will be given, too. Afterwards, we will refer to the possibility of choosing the number of outgoing edges of MDDs used by EXACT. Finally, there is a section considering handling of the software.

2.4.1 Data structures

Internal representation of MDD nodes

The experimental results that are shown in [5] were evaluated by using a prototype MDD package [21]. In order to improve efficiency, we included the state-of-the-art BDD[6] package CUDD [22]. But since we need MDDs instead of BDDs it was necessary to implement a class that "merges" a certain number of BDD nodes [23]. So the objects belonging to this class represent one MDD node each. Figure 2.4 illustrates this for an MDD node with 4 outgoing edges.

Due to this management our class called MVDD includes the following two members:

- `Ddnode *m`:
  this is a pointer to the top variable of the MDD that is represented by the object. Actually, m is a pointer to a BDD node of the underlying CUDD package. But with our MDD node management, a pointer to an MDD variable v is the same as a pointer to the BDD variable that is the lowest with respect to variable ordering among those that belong to the set of BDD variables that are used to represent v. This is shown in Figure 2.4, too.

[5] **M**ulti-Valued **D**ecision **D**iagram.
[6] **B**inary **D**ecision **D**iagram.

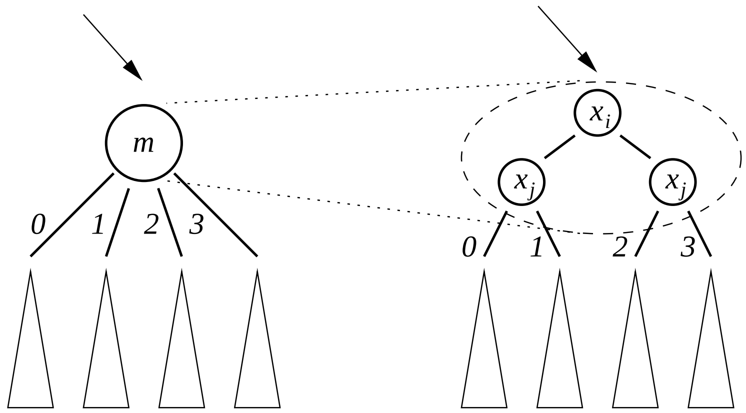


Figure 2.4: internal representation of an MDD node

- `static DdManager manager`:
  For efficiency, DD packages need to include a node manager that takes care about memory management. In our software, we directly use the DD manager provided by `CUDD`. By this, the administration of BDD nodes is done and since MDD nodes are just composed of BDD nodes handling of MDD nodes is implicitly done, too. There is one `manager` shared among all the existing objects of class `MVDD`.

Our goal was exact routing and not to provide a fully featured MDD package based on the `CUDD` package. Therefore, only those functions which are needed for the routing process are implemented as members of class `MVDD`. Substantially, these functions deal with generation of new MDD nodes and logic constants, assignment of leaf nodes to sons of an MDD node and logic AND- resp. OR- operation.

In the following we will give a brief description of the interfaces and the purpose of the member functions of this class:

- `void MVDD_ref()`:
  increments the reference counter for an MDD node.

- `void MVDD_free()`:
  decrements the reference counter for an MDD node.

- `void mk_m(const DdNode& g)`:
  assigns a BDD node `g` to the class member variable `m`. By this, class-external access to `m` is possible.

- `DdNode* dest_m() const`:
  returns the BDD node stored in `m`.

- `static MVDD MVDD_zero()`:
  returns an object of class `MVDD` for the logic constant `0`.

- `static MVDD MVDD_one()`:
  returns an object of class `MVDD` for the logic constant `1`.

- `int MVDD_one_variable_true(int n)`:
  generates an MDD variable with one outgoing edge leading to the logic constant `1` and all the others leading to the logic constant `0`. When calling this function, the following precondition must be fulfilled: there has to be exactly one path from the tree's root leading to leaf `1` and this path has to contain exactly as many inner BDD nodes as there are needed for one MDD node.

- `void MVDD_var_for_field(int i, int j, int k, int n)`:
  creates the BDD variables that are needed for one new MDD variable. A conjunction of these BDD variables is stored in the structure `field` (sec.2.4.1). The parameters don't have any meaning, they remain only for reasons of compatibility to earlier versions that used a different MDD package.

- `MVDD MVDD_and(MVDD g) const`:
  returns the MDD that is the result of a logical AND operation with MDD `g`.

- `MVDD MVDD_or(MVDD g) const`:
  returns the MDD that is the result of a logical OR operation with MDD `g`.

- `static long MVDD_noNodes()`:
  directly returns the number of shared BDD nodes actually used by `CUDD` package.

- `static unsigned MVDD_deadNodes()`:
  returns the number of BDD nodes encountered by `CUDD` package.

- `static long MVDD_peakNoNodes()`:
  returns the peak size number of BDD nodes used by `CUDD` package during computation.

- `static void MVDD_nodeStatistic(char *output)`:
  prints information given by the three functions described above on screen. `output` is a message that can describe the situation in which the function is called and that is printed on screen, too.

- `void MVDD_print_MVDD() const`:
  calls the corresponding `CUDD` function and prints minterms for MDD on screen.

Additionally, there exist overloaded C++ operators for "=","==" and "!=".

Organization of routing process implementation

Besides `MVDD`, `EXACT` contains another class that is called `EXACT`, too. There is a member function `route()` in this class that performs the core algorithm and that is called by `main()`. We will describe the class members and the member functions of this class in the same way we did it for the class `MVDD`.

Again, we start with the class members:

- `int length`:
  denotes the number of columns of the channel for which routing has to be performed. Columns are enumerated from 0 to `length`-1.

- `int height`:
  is the number of tracks of the channel for which routing has to be performed. Tracks are enumerated from 0 to `height`-1.

- `int noLayers`:
  gives the number of layers of the channel for which routing has to be performed. Layers are enumerated from 0 to `noLayers`-1.

- `int noNets`:
  denotes the number of different nets that have to be connected. Nets are enumerated from 1 to `noNets`.

- `int* upperPins`:
  contains the necessary information about pin location in track `height`-1. At position x, there is a 0 if no pin is located in column x, and the number of the net otherwise. This is illustrated in Figure 2.5. Note that no layer unless layer 0 may contain any pins.

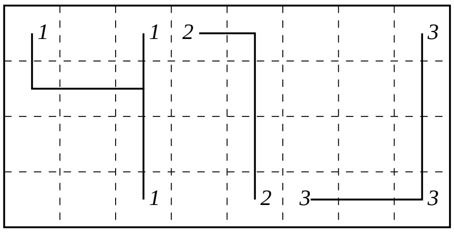
- `int* lowerPins`:
  contains the necessary information about pin location in track 0. At position x, there is a 0 if no pin is located in column x, and the number of the net otherwise. Again, this is shown in Figure 2.5.

- `MVDD*** field`:
  for every cell c in the channel, there is one MDD in `field`. There is only one inner node in such an MDD and the according variable is the one that corresponds to c. One of the outgoing edges of this node leads to leaf `1`, all the others lead to the `0` leaf.

- `MVDD solutions`:
  the MDD in which the results of routing are stored. After routing is finished, one can find the set of *all* possible solutions to the specific CRP.

`upperPins:`

1	*0*	*1*	*2*	*0*	*0*	*0*	*3*



`lowerPins:`

0	*0*	*1*	*0*	*2*	*3*	*0*	*3*

Figure 2.5: data structure for pin location

Now, we will explain the purpose of the class member functions. The `mk_...` and `dest_...` functions enable read resp. write access on class member variables from functions that don't belong to this class. They are straightforward, and for this, they will be omitted in the following detailed description:

- `void init_pins()`:
  ensures that by default there are no pin entries.

- `void init_field()`:
  allocates memory for new MDD variables and creates these variables afterwards.

- `int read_channel(const char *file)`:
  reads the channel from file `../cio/<filename>.cio`. Return value is the read status: 0 for "okay", 1 otherwise.

- `int num_pins(bool upper)`:
  counts number of pins on top resp. bottom of channel and returns this value.

- `void comp_noNets()`:
  finds out how many different nets there are in the channel.

- `route()`:
  "heart" of this class. Computation of routing solutions is completely done here. For each net, connectivity predicates are computed by fixed point iteration.

2.4.2 Number of outgoing edges for MDD nodes

The number of nets that exist in a given channel determines the number of possible values for an MDD variable and thus the number of outgoing edges per MDD node that is needed in this context.

In our MDD management concept, the number of outgoing edges per node is fixed. To be more precise, it is determined by the number of BDD variables that internally represent one MDD variable. n BDD variables can have 2^n different combinations of values and for this an MDD variable represented by n BDD variables has 2^n outgoing edges.

If there are more outgoing edges than required for routing, some of them remain unused and routing works anyway. However, it is not advisable to use MDDs with much more outgoing edges than needed. The reason for this is that even if MDD size remains the same, the underlying BDDs grow considerably when the number of BDD variables representing one MDD variable increases. Extraction of the essential parts of the results also gets more difficult in those cases.

Doing correct exact routing for a channel using MDDs with less outgoing edges per node than required, of course is impossible. Since the number of outgoing edges of MDD nodes is always a power of 2, the optimal solution for a channel with n different nets is to use MDDs with

$$s = \min\{i \in I\!N | i = 2^j, j \in I\!N \wedge i \geq n + 1\}$$

outgoing edges. s has to be greater or equal $n + 1$ because one value is needed to express that a cell in the channel is not occupied by any net.

Up to now, `EXACT` provides `MVDD` classes with 4, 8 and 16 outgoing edges per MDD node. If another class, e.g. one with 32 outgoing edges per node, is needed, the appropriate `MVDD` class file has to be generated and included instead of those mentioned above.

To generate such a file, only the value of the macro `INTERNAL_VARIABLES`[7], the function `MVDD_var_for_field` and the `switch` statement in the function `MVDD_one_variable_true` have to be changed. The change of `MVDD_var_for_field` is straightforward while in `MVDD_var_for_field`, all the possible different combinations of the internal variables have to be considered and inserted.

2.4.3 User manual

The names of the different executables with respect to the different numbers of outgoing edges per MDD node are called `exact4`, `exact8` and `exact16`. Some other executables, e.g. `exact32` can easily be generated by inserting the appropriate rule in `Makefile` if the required source file `mvdd_cudd32.cc` is available.

To start `exactX`[8] the channel for routing has to be given in `cio` format (Section 2.1.2) in a file with extension `.cio`. The MDD representing the solutions is generated and

[7] `INTERNAL_VARIABLES` determines how many BDD variables are used to represent one MDD variable.

[8] This means one of `exact4`, `exact8`, or `exact16`.

printed on screen. Afterwards, some statistical information about MDD resp. BDD size as well as the time needed for computation is shown. Below, the screen output produced by EXACT during routing for the channel easy3 is listed:

```
Exact Routing for Channel 'easy3.cio':
Set of all solutions as MDD minterms:
0010100111000010100111000010100111 00 1

After all computation is done:
Number of nodes - peak size:  8176
Number of dead nodes:  7215
Number of nodes of output:  70
There are 70 non-zero DD reference counts.

After solution is deleted:
Number of nodes - peak size:  8176
Number of dead nodes:  7248
Number of nodes of output:  37
There are 37 non-zero DD reference counts.

Total Duration:  0.04 sec.
---------------------------------------------------------------
```

The command line must have the following format:

```
exactX [--help]|[inputfile][-r][-v]
```

Description of parameters:

--help: shows a short description of the software and does not perform any other steps.

inputfile: determines the filename (without the extension .cio) of the file the channel data is read from. Default is in.

-r: the MDD that represents the set of solutions to the CRP is not printed on screen if this option is used. This may be desirable if only the size of the MDD is of interest.

-v: if this option is used, some additional information is printed on screen during the routing process. Useful for debugging.

In the cio directory there are samples given in several .cio files. Among them are the examples unsolvable, solvable and easy that are mentioned in [5]. For example, the filename solvable5.cio means that this file provides a cio description for the channel solvable with 6 tracks. The number that is part of the filename string denotes the maximum track index and since the minimum index is 0 this number has to be increased by 1 in order to reach the total track number.

3 GAME — Genetic Algorithm Managing Environment

The optimization environment based on *Genetic Algorithms* (GAs) is presented in this chapter. It is used in the project *Routing with Genetic Algorithms* as a basic support. First, the idea behind this software environment is briefly described. Then an example of programming code is given to show how the evolutionary library can be used for optimization in this field. Finally a tool to map GA-based optimizations on parallel computer systems is presented.

3.1 Basic concepts

`GAME` [24] is a software environment for using GAs in the area of Computer Aided Design of Integrated Circuits. Many successful applications of GAs in this domain have been reported in the last few years (for an overview see [25]). The problems dealt with in this field consist of sequences of subproblems, which are characterized by being NP-hard, large and mutually dependent. Multiple competing criteria have to be optimized subject to a large number of non-trivial constraints. This very high level of complexity should make the GA a well suited approach, at least in principle.

Due to the requests of our application, the modularity of the implementation is one of the key issues. This modularity is supported by a standardized *C++* interface used throughout the whole system. Furthermore, the integration of existing software tools and software reuse in `GAME` is an important aspect. Thus an interface to the routing data structure is given, and the integration of standard routing heuristics in `GAME` is supported. Notice, that it needs communication with these tools, but a direct access to the other data structures, i.e. to the routing library, is not needed.

In the past several applications that make use of `GAME` as the underlying optimization tool have been considered. Especially applications in the area of *logic synthesis*, *testing* and *routing* have been successfully presented [26, 20, 4, 27, 28, 29].

3.1.1 Hierarchical class concept

For most real-world applications and especially in the area of routing VLSI systems it is very important to have the flexibility of including arbitrary data structures as individuals in a population instead of string representations only.

To provide this flexibility `GAME` makes use of a hierachical class concept. For a better understanding of the class concept it is first described on an abstract level.

Population class
The kernel of `GAME` is given by a class that handles the populations, the so-called *population manager*. The access to the present population and offsprings is managed, and also both selection schemes and deletion schemes are included at this software level. A template is used to identify the individuals, such that an arbitrary data structure can be handled by the population manager. (An example is given in section 3.2.2.)

Data structure classes
The data structures that are available in `GAME` are specified in several specific classes. They determine the storage of the individuals. Classes for the representation of strings, graphs, trees (*Genetic Programming* (GP) [30]), sets and several problem specific data structures for applications like channel routing are available in `GAME`. All these classes can be used as a template for the *population manager*.

Notice, that for representing the string data structure there historically exists a separated library in `GAME` that does not make use of the population manager and that is not included in the hierarchical system. It is the "classical" GA tool and has been implemented first.

Operator classes
For each data structure class a library supports genetic operators and/or problem specific operators that belong to the chosen data structure.

Based on the structure of this tool the hierarchical software environment `GAME` has been developed.

3.1.2 Genetic operators and integration of heuristics

In the routing tool efficient and state-of-the-art heuristics are provided. Therefore, it must be easy to integrate "external" heuristics in the GA operators. The existing heuristics are available as library functions of the routing system. In `GAME` these problem specific heuristics can directly be used, like genetic operators.

3.1.3 Multi-objective optimization problems

GAs are well suited for solving multi-objective optimization problems where mutually dependent subproblems are considered in parallel. One problem that arises when using GAs is to rank a pool of solutions [31]. In [32] it is described how this is done in `GAME`. The main idea is to sort the solutions into so-called *Satisfiability Classes* (SCs). Depending on the SC of an individual a single fitness value is then assigned. It is also possible to handle constraints and priorities of the objectives during the optimization process.

3.2 User Manual

In the following the core object classes of `GAME` are described. On the one hand, there is the elementary GA-tool for optimizing the string data structure (`class population`) and on the other hand, the template `class pop<chrom>`.

For both libraries it is possible to make use of the feature for handling multi-objective optimization problems (see function `ga_multobj()` in the following subsection).

3.2.1 The library for string representations

The GA library for string representations of `GAME` provides the management of string individuals in one separated class. It can be used for minimization and maximization problems. The default setting corresponds to maximization problems. In the following only some necessary member functions are described (for a detailed description and an overview on existing member function see `$GAME_HOME/incl/game.h`). For using the library for string representations the file `$GAME_HOME/incl/game.h` has to be included by `#include<game.h>` at the beginning of the program file.

For the initialization of an element of `class population` initialization functions are provided. The main objective is to set basic parameters for the GA environment. The mainly used initialization function is explained in the following:

- `init_pop(int chrom_size, int no_of_children, int pop_size, int alphabet_size, char permutation=0, int no_of_mul =1, int *priorities=NULL)`:
  The meaning of the parameters is as follows: `pop_size`, `no_of_children`, `chrom_size` denotes the size of the population in each generation, number of offsprings to be created in each generation and the length of the chromosoms (strings), respectively. `alphabet_size` specifies the size of the alphabet used, i.e. the number of different symbols used in the string representation. If the argument `permutation` is equal to 1, sequencing problems are considered. The default setting is 0, which means that each string of specified length over the given alphabet is allowed as a chromosome. If multi-objective optimization problems are considered, this parameter has to be used. A priority vector can be specified. An integer number at position `priority[`i`]`, $1 \leq i \leq$ `chrom_size` is used to set the priority of objective i. The smaller the integer number is the higher is the priority of that objective. Different objectives may have the same priority.

The program code of a basic GA below shows how to use the GA library of `GAME` for string representations:

```
ga_basic(){
   population *P ;
```

```
    P = new population() ;
    P→init_pop(20,25,50,2) ;
    for (i = 0; i <P→pop_size(); i++ ) {
        long value = evaluate(P→indiv(i)) ;
        P→insert_fitness(P→indiv(i),value) ;
    }
    do {
        P → reproduction() ;
        for (i = 0; i < P →pop_size(); i++) {
            value = evaluate(P→indiv(i)) ;
            P→insert_fitness(P→indiv(i),value) ;
        }
        P→update_population() ;
    } while (improvement_obtained) ;
    return P→get_best_element() ;
}
```

The population `P` handles strings of length 20 and the size of the population is set to 50. In each generation 25 children are created, and the size of the alphabet is set to 2. These settings are specified by function `init_pop(20,25,50,2)`. Member function `void reproduction()` is used to produce offsprings. It makes use of genetic operators and selection mechanisms, which are also member functions of `class population`. The initially generated and the newly created individuals are evaluated by an user defined function `long evaluate(chrom)`. It calculates the fitness of an element in a population. Then the fitness for each element is stored by the member function `void insert_fitness(chrom,long)`. The individuals of type `chrom` can be enumerated by the member function `chrom indiv(int)`.

The user can also encode a reproduction routine by himself using the genetic and selection operators. Function `void update_population()` inserts the newly created individuals in the population an deletes the worse ones. The `while`-loop is repeated, until a termination criterion holds, and finally the best element is returned.

In function `ga_multobj()` it is shown how to use `GAME`, if multi-objective optimization problems are considered:

```
ga_mulobj() {
    population *P ;
    P = new population() ;
    P→init_pop(20,25,50,2,0,k,PRIORITIES) ;
    for (i = 0; i <P→pop_size(); i++ ) {
        long value = evaluate_obj1(P→indiv(i)) ;
        P→insert_rawfit(P→indiv(i),0,value) ;
        value = evaluate_obj2(P→indiv(i)) ;
```

```
        P→insert_rawfit(P→indiv(i),1,value) ;
           ...
        value = evaluate_objk(P→indiv(i)) ;
        P→insert_rawfit(P→indiv(i),k − 1,value) ;
     }
     do {
        P → reproduction() ;
        for (i = 0; i <P→pop_size(); i + +) {
           value = evaluate_obj1(P→indiv(i)) ;
           P→insert_rawfit(P→indiv(i),0,value) ;
           value = evaluate_obj2(P→indiv(i)) ;
           P→insert_rawfit(P→indiv(i),1,value) ;
           ...
           value = evaluate_objk(P→indiv(i)) ;
           P→insert_rawfit(P→indiv(i),k − 1,value) ;
        }
        P→update_population() ;
     } while (improvement_obtained) ;
     return P→get_best_element() ;
  }
```

Member function `init_pop(20,25,50,2,0,`k`,PRIORITIES)` sets the parameter, such that `GAME` is able to optimize problems with k optimization goals. Priorities are set by the vector `int *PRIORITIES`. `PRIORITIES` has length k and `PRIORITIES[i]` gives the priority of objective $i, 1 \leq i \leq k$. The evaluation process of an individual now calculates the fitness of each objective. It is then stored by function `void insert_rawfit(chrom,int,long)`, where the first argument denotes the considered individual, the second argument denotes which objective is stored (enumerated from 0 to $k-1$), and the third argument is the fitness value to be inserted[1]. No further interaction by the user is required.

The reproduction process can be alternatively encoded by the user itself. Then it is also possible to include problem specific heuristics as genetic operators. This works as follows:

```
  my_own_reproduction(P) {
     int one = P→roulette_wheel_selection() ;
     int two = P→roulette_wheel_selection() ;
     chrom indiv1 = P→indiv(one) ;
     chrom indiv2 = P→indiv(two) ;
     for (i = 0; i <P→no_of_children(); i + +) {
```

[1] The name *rawfit* comes from the term *raw fitness* which denotes the “raw” fitness of each objective.

```
            if (random()%2)
                P→crossoverI(indiv1,indiv2) ;
            else {
                heuristic(indiv1,P→child(P→inc_children()-1) ;
                heuristic(indiv2,P→child(P→inc_children()-1) ;
            }
      } ;
```

First, two individuals of type `chrom` are determined by roulette-wheel selection (functions `int roulette-wheel-selection()` and `chrom indiv(int)`). Then the offsprings are generated. The number of offsprings is returned by function `int no_of_children()`. Then with probability 0.5 the two individuals are recombined by function `void crossoverI(chrom,chrom)`, where a crossover of both elements is turned out. Otherwise, both elements are modified by a problem specific heuristics. Notice, that the resulting element after applying operator `void heuristic(chrom,chrom)` has to be stored as an offspring by function `chrom child(int)` at the index given as an argument. Additionally, the present children counter (internal variable) has to be incremented by function `int inc_children()`.

3.2.2 The population manager

In the following the basic member functions of the template `class pop<chrom>` are outlined. `class pop<chrom>` is the population manager of the hierarchical part of `GAME`. For a complete description of the library see the documentation in the program file `$POP_HOME/incl/pop.h`. For using the library `$POP_HOME/incl/pop.h` it has to be included `#include<pop.h>`.

For the initialization of an object of the template `class pop<chrom>` the constructor with several arguments has to be used:

- `pop(unsigned short int pop_size=50, unsigned short int no_of_children=0, unsigned char max_problem=1, unsigned short int chrom_size=` -1`, unsigned int no_of_mult=1, unsigned int *priorities=NULL)`:
  Basic parameters for the GA can be set as follows: `pop_size` (`no_of_children`) specifies the number of elements in the population (to be created). If the parameter `max_problem` is 1, maximization problems are considered, otherwise if it is set to 0 minimization is set. `chrom_size` gives the user the possibility to store a size value for the chromosomes. Again, if the array `priorities` is not NULL, multi-objective problems are considered. The priority vector must have positive valued entries and the user has the possibility to assign priorities to the certain objectives.

The following example shows how to use the template `class pop<chrom>`. The `class data_structure` is an arbitrary data structure that is taken as template by the population manager as individuals:

```
ga_pop() ; {
   pop<data_structure> *P ;
   P = new pop<data_structure>(20,10) ;
   init_pop(P) ;
   for (i = 0; i <P→pop_size(); i++ ) {
      double value = evaluate(P→indiv(i)) ;
      P→insert_fitness(i,value) ;
   }
   do{
     create_children(P) ;
     for (i = 0; i <P→pop_size(); i++ ) {
         value = evaluate(P→indiv(i)) ;
         P→insert_fitness(i,value) ;
     }
     P→update_population() ;
   } while (improvement_obtained) ;
   return P→get_best_chrom() ;
}
```

In this program the population size is set to 20 and in each generation 10 new individuals are created (see constructor of `class pop<chrom>`). The functions `void init_pop(class pop<chrom>)` and `void evaluate(class pop<chrom>)` are dependent on `class data_structure` and thus they have to be specified by the user. Function `void create_children(class pop<chrom>)` manipulates elements of type `class data_structure` and thus it also has to be specified by the user. It can make use of member functions from `class pop<chrom>`, e.g. the selection mechanisms. The management of the population and offsprings is done by member function `void update_population()`.

Finally, it should be mentioned that problem specific data structure classes and operator classes have been implemented.

3.3 GA-based optimization on parallel computer systems

`EPA` [33] is a software tool for mapping GAs and GPs onto parallel computer systems. It allows the engineer to define arbitrary topologies of the communication net of the processors. All the features are controlled by a script which is specified by the user. This script consists of three parts

```
definitions
%%
EPA functions
%%
user specific routines
```

which contain all information necessary to run the GA or the GP, including the class of the individuals which compose the populations, the user specific genetic operators, and the procedures to pack and unpack the information to be transfered between the processors. Such a script whose syntax looks like that of `Lex` and `Yacc` is translated into a C++ program extended by `MPI` (Message Passing Interface) routines.

In the following we will present the three parts of an `EPA` script in more detail. For illustration, we look at the problem of finding an assignment to the variables x, y, and z such that the function $f(x, y, z) = |x - y| + |y - z| + |z - x|$ is minimized.

3.3.1 Definitions

The definition part has to be written in C++. It consists of the definitions of the types, constants, and variables needed. In particular, the class definition of the individuals of the populations can be given in this part of the script. This definition has to be marked by the symbol `%i`.

```
%i
class individ
{
   int g[3];
  public:
   int get_gen(int k)
   { return ( g[k] )};
   ...
   double fitness()
   { return ( abs(g[0]-g[1])+abs(g[1]-g[2])+abs(g[2]-g[3]) ); };
   ...
   void pack(char *buffer)
   sprintf (buffer,"%d %d %d",g[0],g[1],g[2]); };
   ...
   void unpack(char *buffer)
   sscanf (buffer,"%d %d %d",&g[0],&g[1],&g[2]);
}
```

`pack` and `unpack` denote the functions to pack and unpack the information in order to be sent from one processor to another one.

3.3.2 EPA functions

This part allows the user to redefine the communication topology of the processors, to define the sizes of the populations located at different processors which may vary from generation to generation, and the probabilities of mutation and cross over, and so on.

Before presenting the main `EPA functions` let us illustrate this part of the script by example

```
%%
epa_pop_size = 20
epa_generations = 500
epa_selection = 1

/* communication topology is a binary tree
(bottom up communication )*/
for ($(i=31); $(i>1); i++)
    if ($(i%2)==0)
    { epasend $((i-1))%2) -b }
    else
    { epasend $(i%2) -b }
```

As illustrated in this example, process i is denoted by `$i`. In order to address all processes, the symbol `$#` is used. Generation j can be addressed by symbol `@j`. To address all the generations, we use symbol `@#`.

In the example, the initial population size for each process is set to 20 individuals. The number of generations is set to 500. As selection method, roulette wheel is taken (`epa_selection=1`). The topology of the virtual parallel machine is given by a binary tree in which each node only communicates with its father. Option `-b` determines that the best individuals have to be sent to the communication partner.

The following `EPA` functions and variables are available, in particular:

- `epa_population`
  specifies the size of the population of the current process.

- `epa_generations`
  specifies the number of generations.

- `epa_selection`
  specifies the selection method. Value 1 denotes the roulette wheel method, value 2 random selection, and value 3 a user specific selection method.

- `epa_pc`
  specifies the probability for cross over.

- `epa_pm`
  specifies the probability for mutation.
- `epa_send_number`
  specifies the number of individuals to be sent to neighbored processes.
- `epaexit`
  stops the corresponding process.
- `epasleep`
  inactivates the corresponding process.
- `epastart`
  wakes up the process which has been inactive because of an `epasleep` command.
- `epasave [ all | M-N ] file`
  saves all the individuals or the individuals located at the positions `M`, `M+1`, ..., `N` of the current population. Note that the individuals are ordered with respect to their fitness.
- `epasave_best N file`
  saves the `N` best individuals of the current populations of the parallel system.
- `epaload [-b | -w | -r] M file`
  loads `M` individuals into the current population. The options `-b`, `-w`, and `-r` determines that the `M` best individuals, the `M` worse individuals, or `M` random individuals are substituted by the individuals loaded.
- `epasend [-b | -w | -r] [ {process}`$^+$ `| all]`
  determines the processes with which the process communicates. The options `-b`, `-w`, and `-r` determines that the `M` best individuals, the `M` worse individuals, or `M` random individuals are to be sent.

3.3.3 User specific routines

This part contains user specific functions, e.g., mutation / crossover operators or functions gathering statistical information.

4 Heuristic Learning

We briefly describe the idea of the *Heuristic Learning* (HL) approach and show how `GAME` and `MaDRE` interact for its software realization. A more detailed description of HL can be found in [34], successful applications are reported in [35, 36] Afterwards, the optimization criteria that are used for HL are explained followed by a description of important functions of the software introduced in this chapter. We explain the correct interpretation of the information provided by a special log file that is generated while evolution in the GA is going on. Finally, parameters and their meaning are described in the user manual.

4.1 The approach

In many cases, differing non-optimal strategies have to be applied to solve certain instances of specific optimization problems; on the other hand, these strategies fail for other classes of benchmarks. The idea behind HL now is to combine these strategies in a way that advantages of the differing methods can be utilized and thus more problems can be solved satisfactorily by the newly generated heuristics. The combination of the strategies is done by using a GA to *learn* a heuristics on a so-called training set. Thus, the learning phase, which sometimes may be costly, is transferred to a preprocessing step and the application of the heuristics itself usually is very time-efficient.

In our context, different strategies resp. routers are considered as *Basic Optimization Modules* (BOMs). A heustistic is a sequence of BOMs. Solving a specific problem instance, i.e. in our case performing routing for a given channel with a heuristics, means sequential application of the BOMs to this channel. BOM $i+1$ starts with the prerouted channel, i.e. with a channel containing all the wires that have been established by the BOMs $1 \ldots i$ before. In the GA which we use to learn heuristics, individuals represent heuristics and their gene strings represent ordered lists of the BOMs the heuristics consists of. That is, in gene i, the number of the ith router to be applied to the benchmark channel is given.

Figure 4.1 illustrates the concept of HL. The set of BOMs consists of several routing stategies[1]. The GA selects some of them for a heuristics that is applied to the benchmark shown in the figure. Routing for this benchmark is split up into several

[1] We use the strategies that have been introduced in Chapter 2.

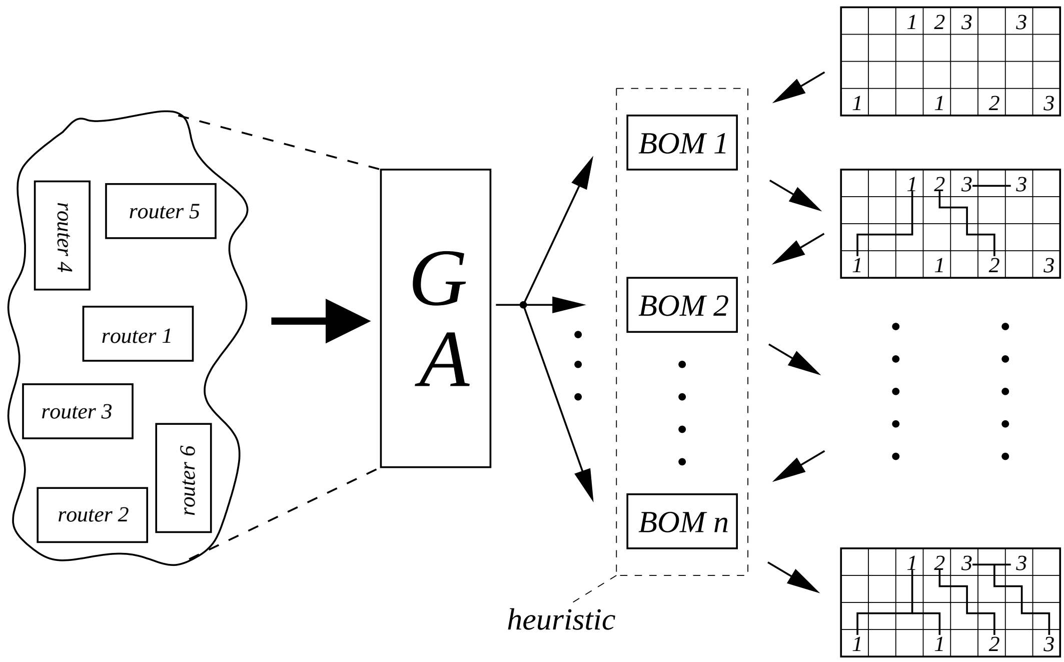


Figure 4.1: Scheme for Heuristic Learning for channel routing.

steps and the ith step is performed by the router that has been selected as BOM i for the heuristics.

4.2 Interaction of software libraries

We have already mentioned in Chapter 1, that the tools introduced in Chapter 3 and 2 are intensively used by our HL software.

`GAME` provides the data stucture that is needed for the GA and that has already been mentioned in 3.1. During evolution, this data structure contains all the essential information about the population including individuals or the corresponding gene strings, respectively.

Additionally, many fundamental functions needed in a GA are directly taken from the `GAME` environment. Section 4.4 gives a more detailed description about the `GAME` functions we used.

Finally, our software also makes use of the possibility for multiple-objective optimization provided by `GAME`. Our optimization criteria for routing are considered in Section 4.3.

As well-known, in GAs individuals have to be evaluated during evolution for many times. The function that evaluates the fitness of a given individual is dependent on the specific optimization problem to which HL is applied. Therefore, this evaluation function can not be provided by `GAME` (see Section 3.2).

For evaluation of an individual, routing has to be applied to the benchmark channels of the training set in the way it is shown in Figure 4.1. That is, the routing strategies that correspond to the BOMs have to be applied to the given channel in sequential order.

Therefore, the routers and the data structure included in `MaDRE` can be used as follows[2]: at the beginning of the evolution, an integer identifier is assigned to each instance of a router that is used as BOM. Different identifiers are assigned to BOMs even if they only differ by truncation criteria. The gene strings of the individuals contain only these identifiers. For computation of the fitness of an individual, several steps have to be performed. First of all, the data of a benchmark channel[3] has to be read from a file into the internal `MaDRE` data structure `TASK` (see Section 2.1.1). Then, the identifiers in the gene strings are considered in sequential order and the corresponding routing strategies are consecutively applied to the channel by calling the appropriate function of the `MaDRE` environment. In that way, the wires that have been generated by the router corresponding to BOM i remain in the channel resp. the data structure `TASK` when calling the router that corresponds to BOM $i+1$. By this, routing is done with respect to parts of the channel that have already been routed before.

When routing is finished, the properties of the channel have to be computed. This can be done by simply calling the `compute_fitness()` function that belongs to `MaDRE` and is described in Section 2.1.3. Finally, the appropriate values for the optimization criteria can be taken from the `quality` data structure. These optimization criteria will be explained in the following section.

4.3 Optimization criteria

Different from many other optimization problems, for routing it is desirable not only to minimize or maximize one single criterion but to optimize several properties of routed channels in parallel. These optimization criteria are mutually dependent, what makes parallel optimization even harder. `GAME` provides the possiblity to perform such multi-objective optimization (see Section 3.1) and even priority handling is possible.

We make use of this technique and therefore all of the applications that are introduced in Section 4.4 minimize the following 5 criteria in parallel:

- number of nets that are not completely routed.
- number of vias, i.e. layer changes in the channel.

[2] Note that more subtle encodings will be used in future

[3] This channel *may*, but *does not require* to contain any prerouted wires.

- overall number of wire segments needed for the connections.
- number of tracks the channel contains.
- time in 1/100 seconds that was needed for application of a specific heuristics to the benchmark channel.

The routers introduced in Sections 2.3.1–2.3.4 may increase the number of tracks in the channel while building up the connections for the nets. Obviously, the area needed for the channel is also increased in that way. Of course, we want to keep area consumption as low as possible, and for this, one of our goals is to minimize the number of tracks in the channel.

The remaining routing strategies described in Chapter 2, i.e., the lineprobe router, the maze routers, and the random router, are not able to change the number of tracks in the channel. When a heuristics exclusively consists of such routers, the number of tracks still remains an optimization criterion. But since it is constant in these cases, it has no influence on the optimization process.

We have already mentioned that these criteria are not independent from each other. E.g., if the number of established connections decreases and therefore the number of nets not completely connected increases, of course the number of vias and also the number of wire segments decreases. Thus, the values for this optimization criteria get better.

A specific optimization criterion may be more or less significant than another one. Thus, when using `GAME`, it is possible to assign different priorities to the optimization criteria. We use the priority values described in the table below. A lower priority value means a higher priority.

optimization criterion	priority
number of not connected nets	1
number of layer changes	2
total number of wire segments	2
number of tracks in the channel	2
time consumption for routing	3

Among the optimization criteria, the number of completely connected nets is the most important one. If for two channels, the numbers of completely connected nets are equal, we usually do not consider one of the criteria "number of vias", "total wirelength" and "number of tracks" more important than any of the other two of these criteria. Thus we assign the same priority to all of them. Time consumption for application of a specific heuristics is not very critical. So the corresponding criterion is used for fine tuning of the heuristics, if and only if all the values for the other criteria are equal for two different heuristics.

4.4 Software components for Heuristic Learning

Currently we are working with two components, HL_ROUT and RR_HL, to perform HL. While the set of BOMs for HL_ROUT consists of the routers introduced in Sections 2.3.1–2.3.4 with each of them having a fixed combination of parameter settings, the BOMs used in RR_HL are represented by 10 different instances of the random router. The difference between these instances are the settings for the fitness values described in Section 2.3.8 and the initialization value for the random number generator. Especially the settings of the fitness values influence the behaviour of the random router considerably. Both the sets of BOMs for HL_ROUT and RR_HL, respectively, contain one additional element: the empty operation. By using this BOM, we include an implicit optimization of the number of BOMs used in a heuristics. E.g., if the optimal heuristics contains 5 BOMs and we use a gene string consisting of 8 genes with each of them representing one BOM, we suppose our approach to generate the optimal sequence of 5 BOMs that represent specific routers with three additional "0" entries somewhere in between.

Furthermore, RR_PLERN that does not deal with HL, but nevertheless is very similar to HL_ROUT and RR_HL from the technical point of view has been implemented, too. This tool also uses a GA, but the goal is not to generate a well suited combination of instances of single routers, but to perform evolution in order to find an optimal set of parameter values for the random router. Due to its close relationship to the other programs, it is described in this chapter, too, although it does not perform HL (in the strong sense).

In the following sections, the functions that realize the fundamental parts of the GA in HL_ROUT, RR_HL and RR_PLERN are described. The functions that deal with the evolution loop and the generation of offsprings are nearly the same for the three components, and so they are described only once. The evaluation of individuals is different for the three approaches and therefore the corresponding functions are described separately.

4.4.1 Evolution

The function `genAlg()` realizes the evolution loop for the GA. First, some initalization is done. The number of offsprings that are created in each generation is computed followed by the initialization of the population. Afterwards, the individuals of the initial population are evaluated, i.e., their fitness is computed. The individuals are ordered with respect to their fitness by using the corresponding ordering algorithm in the multi-objective optimization part of GAME. After these steps evolution starts. The offsprings for the next generation are generated (see Section 4.4.2) and evaluated. Selection of parents is considered as part of the offspring generation and therefore described in Section 4.4.2, too. If any of the created offsprings are duplicates of already existing individuals, they are mutated once more and evaluated again. Now, the "survival" step is performed. The last essential step of the loop is a reordering of the individuals with respect to their fitness that is necessary due to the fact that the offsprings have been merged into the population before. The loop is stopped if there has been no improvement for a specific number of consecutively computed generations and it is repeated otherwise.

In the following, we list the most important `GAME` functions that are used by `genAlg()` and give a short description of their purpose:

- `init_pop(`*ldots*`)`:
  generates the initial population.
- `compute_ranking()`:
  creates an ordering of the individuals with respect to their fitness.
- `index_sort()`:
  sorts the indices of the individuals in population in an internal data structure with respect to their fitness.
- `delete_n_last()`:
  the survival function. Simply replaces the worst individuals by the previously generated offsprings.
- `insert_fitness(...)`:
  after computing the new ranking, an overall fitness value is assigned to each of the individuals.
- `best_generation()`:
  returns the number of generations since the last improvement of the best individual in the population has occured.

4.4.2 Generation of offsprings

The purpose of the function `next_generation(...)` is to select parents and create offsprings for the next generation during evolution.

This is done as follows: the offsprings are generated sequentially. For each of these steps, at first two different parents are selected by the so-called *tournament selection*. Then, according to some given probabilities, crossover, mutation or inversion operations are performed. To do so, one out of many possible mutation resp. crossover operators is used. Finally, the resulting offsprings are mutated once again.

4.4.3 Evaluation of fitness

In Section 4.2 we have alrady explained how evaluation of individuals theoretically works for HL. In this section, we describe the functions that realize the evaluation in `HL_ROUT`, `RR_HL`, and `RR_PLERN`.

HL with different routers

For evaluation of an individual, several variables representing all the parameters that are needed by the routing strategies representing the BOMs have to be defined first. Then, the channel data for the benchmark is read from the input file.

Afterwards, sequentially the genes of the individual that has to be evaluated are read, decoded and the corresponding routing strategies are applied to the instance of the benchmark channel given by the data structure `TASK`.

`MANHATTAN` is used as routing model here, overlap model is `VH`. In addition to this, the parameters `test_density_viol` and also `test_macle_viol` are set to `TRUE` for all BOMs. Further parameter settings shared by all BOMs are `min_used_track` that is set to 0 and `s_typ` that is set to `column`. Additionally, only algorithm 1 of the Yoshimura&Kuh router is used.

The rest of the parameter settings is different for the single routers and therefore is shown together with the decoding scheme in the following table:

index of BOM	routing strategy	parameters				
		rs	range	yndl	ydl	rdl
0	empty operation	-	-	-	-	-
1	constraint leftedge router	TL	-	-	-	-
2	dogleg router	TL	4	-	-	-
3	yoshimura router	T	2	TRUE	FALSE	FALSE
4	yoshimura router	T_B	3	FALSE	FALSE	TRUE
5	yoshimura & kuh router	-	1	TRUE	FALSE	FALSE
6	yoshimura & kuh router	-	3	FALSE	FALSE	TRUE

In the table, some abreviations are used: “rs” for `routing sequence`, “yndl” for `yk_nondogleg`, “ydl” for `yk_dogleg` and finally “rdl” for `reduce_doglegs`.

After all BOMs have been applied to the channel, the fitness is computed by the appropriate function provided by `MaDRE` and afterwards the values for the optimization criteria are put into the data structure for the population.

Figure 4.2 shows an excerpt of the evaluation function.

HL with different instances of the random router

The evaluation function for `RR_HL` works in the same way as the one for `HL_ROUT` previously described, only the set of BOMs and the parameters that have to be set are different. Here, no routers besides the random router are used. We define 5 different sets of fitness values for specific kinds of neighbour points and use them as parameters. Additionally, we combine each of these sets with two different initialization values (2 and 5) for the random number generator. Thus we obtain 10 different instances of the random router and use these as BOMs for HL.

```
long evaluate(population *P, chrom ch, stop_watch *st)
{
  int     routing_model = MANHATTAN;
   ...
  start_typ  s_typ = column;
   ...
  if(!ReadChannel(&t, in_file))
    ga_error("ERROR: reading channel failed !",i);
   ...
  for ( i = 0; i < P->chrom_size(); i++)
    {
      switch( P->get_gen(ch,i))
        {
        case 0: // empty operation
          break;
        case 1:
          overlap_model=VH;
          rs = TL;
          test_density_viol=TRUE;
          test_macle_viol=TRUE;
          LeftEdgeConstr (&t, overlap_model, rs, min_used_track,
                              test_density_viol, test_macle_viol);
          break;
           ...
        case 6:
           ...
          YoshimuraKuh (&t, 1, overlap_model, yk_nondogleg,
                            yk_dogleg, yk_reduced_dogleg, s_typ,
                            range, range2, 100, 100, min_used_track,
                            test_density_viol, test_macle_viol);
          break;
        }
    }
   ...
  compute_fitness(&t);

  failedNets = t.quality.all_nets - t.quality.con_nets;

  P->insert_rawfit(ch, LAYCHANGES , t.quality.all_lcha);
   ...
  P->insert_rawfit(ch, TIMENEEDED , time);
   ...
}
```

Figure 4.2: Excerpt of the evaluation function of HL_ROUT.

The fitness values of the instances are shown in the table below:

Neighbours			Index of BOM				
d	l	m	1	2	3	4	5
-	-	0	1	5	1	1	1
X	-	0	0	3	0	0	0
X	X	0	0	2	0	0	0
-	X	0	0	3	0	0	0
-	-	1	2048	4	1	1	1
X	-	1	0	3	1	0	0
X	X	1	1	2	0	0	0
-	X	1	2048	3	0	0	1
-	-	2	1024	4	1024	1	1
X	-	2	1	3	256	0	1
X	X	2	1	3	1	0	1
-	X	2	128	3	2	0	1

The abreviations used in the table have the following meaning:

d: distance to target increases when connection to neighbour point established (detour).
l: neighbour is located on different layer (layer change).
m: global mode internally used by the random router.

The different BOMs represent the following strategies:

BOM 1: original strategy. Layer changes are allowed in mode 1 or higher, detours are allowed only in mode 2.
BOM 2: allows some more exploration. Only small difference between the fitness values.
BOM 3: prefers detours instead of layer changes. The latter are allowed only in mode 2.
BOM 4: does no exploration. Neither detours nor layer changes are accepted in any case.
BOM 5: is restrictive in mode 0 and 1 but assigns the same probability to all kinds of possible steps in mode 2.

Learning of parameters for the random router

The evaluation function for this approach is similar to the ones for `HL_ROUT` and `RR_HL` that have been described above. Again, the channel data has to be read at the beginning and the fitness values are set at the end when all routing has been finished. But in opposite to the other approaches, we do not apply several instances of routers consecutively to the benchmark. Instead of this we have a much wider variety of parameter settings, determine their values for the random router and then apply it to the benchmark exactly once.

There are 15 parameters that have to be learned. The first 12 of them are the fitness values that have been used in `RR_HL` to create different instances of the random router. Number 13 and 14 are upper bounds for the truncation criteria "detour per connection" resp. "detour per net". The last parameter is the initialization value for the random number generator.

The set of possible values of these parameters are the integer numbers in the range $0 \ldots 3000$. If the value for one of the truncation criteria exceeds 2000, this causes the router to ignore the criterion. The values in the range $0 \ldots 2000$ are linearly mapped to the interval [1,4].

Figure 4.3 shows an excerpt of the evaluation function. In this part the parameter settings are performed and by this means the random router is prepared for its application to the benchmark channel that is performed by the last command shown in the figure. With the explanations that have been given in this section, understanding of the algorithm shown in the figure should be easy.

4.5 The log file

While evolution is going on in a GA, lots of individuals are generated, mutated, evaluated and maybe deleted again after remaining in the population for some generations. Useful information about the actual population is present in the internal data structures of `GAME`. But when the GA stops, we are interested not only in information about the population of the last generation, but also some knowledge about earlier populations may be important. Moreover, there is a need for visualization of the data contained in the internal structure, i.e. the data has to be be prepared for a clear presentation to the user.

For these reasons, we generate a log file in our GA. After initialization of the population and again once in every iteration of the evolution loop, for all individuals contained in the actual population some specific data is written to that file if the according command line parameter is used. This data includes the genes of the individuals' gene strings and the values for the optimization criteria as well as the overall fitness value, i.e. the satisfiability class the individual belongs to.

The messages that the routers create when being applied during evaluation of individuals are printed out on screen. By this, the two different kinds of output produced by the GA are separated since the log data is written to a file.

An excerpt of a typical log file generated by `RR_HL` is illustrated in Figure 4.4. It shows a population that consists of 5 individuals with gene string representing heuristics that are created by combination of 5 BOMs. For each individual of a population, there is exactly one line in the log file. On the left of this line the values

```
long evaluate(population *P, chrom ch,stop_watch *st,bool evolution)
{
  ...
  for ( i = 0; i < noParameters; i++)
    {
      if (i >= DETOURPERCON && i <= DETOURPERNET)
        {
          // mapping of value for truncation criteria
          gen = P->get_gen(ch,i);
          if (gen > 2000)
            truncVal = IgnoreCriterium;
          else
            truncVal = (3 / 2000.0) * gen + 1;
        }
      if (i < DETOURPERCON) // this means, actual parameter is
                            // a fitness value
        {
          // find out mode and table column
          // of the actual fitness value
          indivMode = i / 4;
          indivSort = i \% 4;
          indivSort *= 2;
          if (indivSort == 6)
            indivSort--;
          router.mk_fitVal(indivMode, indivSort, P->get_gen(ch,i));
        }
      else if (i == DETOURPERCON)
        router.mk_abort(DETOUR, B_Con, truncVal);
      else if (i == DETOURPERNET)
        router.mk_abort(DETOUR, B_Net, truncVal);
      else if (i == RANDOMSEED)
        {
          if (!setSeed)
            rand_init = P->get_gen(ch,i);
        }
    }
  router.rand_rout();
   ...
}
```

Figure 4.3: Excerpt of the evaluation function of RR_PLERN.

of the genes are shown. The middle block consists of the values for the optimization criteria, e.g. it can easily be seen that the population after generation 19 contains three heuristics resp. individuals that can connect all the nets while when applying one of the other two heuristics to the benchmark, one net remains unrouted. The satisfiability class of the individuals are given on the rightmost block of each line.

```
 ...

Individuals in generation 19:
gen string          fitness                   satisf. class
 8  7  6  3  9        1   9 117   7  40        2
 4  1  6 10  0        0   8  71   7   8        1
 2  8  6 10  0        1  15  96   7  60        3
 4  1  6  3  9        0   8  71   7   8        1
 4  1  6 10  0        0   8  71   7   8        1

Individuals in generation 20:
gen string         fitness                   satisf. class
 4  1  6 10  9        0   8  71   7  10        1
 4  1  6 10  0        0   8  71   7   8        1
 ...
```

Figure 4.4: Excerpt of a typical log file produced by the GA.

At the beginning of a log file, some global information like encoding of the genes' values, parameter settings commonly shared by all BOMs and also the name of the benchmark channel is given. The purpose of the GA resp. optimization task is mentioned, too.

4.6 User Manual

According to the approaches previously described there are three executables called `hl_rout`, `rr_hl` and `rr_pLern`. With respect to the syntax, all of them provide exactly the same command line parameters. The command line must have the following format:

```
exe [--help]|[inputfile][-b<noBOMs>][-c<preset>]
    [-g<generations>][-l][-la<overlap model>]
    [-la<routing model>][-laNOCHANNEL|CHANNCONV]
    [-o<noOffsprings>][-p<popSize>][-s<seed>]
    [-t<strategy>][-v]
```

`exe` has to be one out of `hl_rout`, `rr_hl` and `rr_pLern`.

Description of parameters:

`--help`: shows a short description of the software and does not perform any other steps.

`inputfile`: determines the filename of the file the data of the benchmark channel is read from. Default is `klein_test.cio`.

`-b<noBOMs>`: determines the number of BOMs that are combined to one heuristics. In parameter learning for the random router, only the first `noBOMs` parameters are considered for optimization if the `-b` parameter is used. Default value is 3.

`-c<preset>`: provides the possibility to merge one specific individual into the start population of the GA.

`-g<generations>`: determines the maximum number of generations that may be computed without any improvement of the best individual in the population before evolution is stopped. Default value is 50.

`-l`: enables generation of a log file called `ga_rout.log`.

`-la<argument>`: chooses routing model, overlap model and/or sets the channel convention flag.

`-o<noOffsprings>`: determines the number of offsprings that are created in each generation. By default, half as many offsprings as there are individuals in the population are created per generation.

`-p<popSize>`: determines the number of individuals contained by the population. Default value is 5.

`-s<seed>`: initializes the random number generator with `seed`. Default value is 1.

`-t<strategy>`: instead of performing the GA the software can be used to evaluate only one single individual by using this parameter.

`-v`: enables verbose mode.

5 Benchmark Channels and Channel Visualization

As mentioned in Section 2.1.2, the CIO file format is used for the interchange of data, i.e., instances of CRPs and (partly) routed channels as well, between routing tools and the users. For testing purposes we have collected a number of benchmarks given in the tables of Section 5.1.

The CIO file format is helpful to define properties of a certain channel exactly. However, it is very difficult to get an impression of routing quality, dense regions of a channel, unnecessary detours or some other features of a routed channel only by having a look at the according `.cio` file. Therefore, there is a requirement for a convenient visualization of a channel. Visualization is discussed in Section 5.2.

5.1 Channel benchmarks

We use the following shortcuts to describe the benchmarks given at the end of this section:

nets: number of nets

col.: number of columns

mcl.: length of longest path in VCG, ∞: denotes a cycle

d-mcl.: length of the longest path if each multi-terminal net is split into two-terminal nets

dens.: density

z.: number of zones

d-z.: number of zones if each multiterminal net is split into two-terminal nets

mp.: maximal number of pins in a net

o.: - no obstacle; + obstacles

ref.: reference for the channel

The benchmarks we use in our experiments are given in the following tables. We have subdivided them in three groups, namely channels taken from literature, channels made available by DResearch GmbH, Berlin, and switch box instances with inner pins from DResearch GmbH. The second group is subdivided again in small and large channels.

Channels from literature										
name	nets	col.	mcl.	d-mcl.	dens.	z.	d-z.	mp.	o.	ref.
yk	10	13	4	4	5	5	7	3	-	[14]
burstein	10	14	∞	11	4	6	10	3	-	[37]
sherwani	8	16	1	1	4	4	4	2	-	[11]
deutsch	72	177	23	9	19	33	136	12	-	[10]
ex3a	44	89	4	4	15	16	39	6	-	[13]
ex3b	47	84	9	7	17	19	46	9	-	[13]
ex3c	54	103	6	6	18	20	51	7	-	[13]
ex4b	54	119	13	6	17	28	88	9	-	[13]
ex5	64	128	3	3	20	35	65	7	-	[13]

Small channels from DResearch GmbH										
name.cio	nets	col.	mcl.	d-mcl.	dens.	z.	d-z.	mp.	o.	ref.
pr1ch0	26	481	2	2	18	6	6	2	-	[38]
pr1ch1	158	480	5	5	69	59	161	12	-	[38]
pr1ch2	177	481	4	3	83	67	143	6	-	[38]
pr1ch3	167	491	5	5	65	73	115	5	-	[38]
pr1ch4	166	486	5	4	72	59	135	8	-	[38]
pr1ch5	195	490	4	4	73	90	152	6	-	[38]
pr1ch6	189	477	5	5	68	85	165	10	-	[38]
pr1ch7	224	487	∞	∞	95	86	196	9	-	[38]
pr1ch8	26	484	1	1	14	9	9	2	-	[38]

Large channels from DResearch GmbH										
name.cio	nets	col.	mcl.	d-mcl.	dens.	z.	d-z.	mp.	o.	ref.
CH0	53	1312	2	2	14	20	20	2	-	[38]
CH1	399	1312	∞	∞	27	237	387	10	-	[38]
CH2	438	1323	∞	∞	27	265	390	9	-	[38]
CH3	545	1329	∞	∞	32	324	432	15	-	[38]
CH4	575	1333	∞	∞	31	366	436	9	-	[38]
CH5	582	1327	∞	∞	30	382	514	12	-	[38]
CH6	650	1342	∞	∞	45	379	459	9	-	[38]
CH7	663	1314	∞	6	38	405	550	21	-	[38]
CH8	678	1309	∞	∞	35	427	496	13	-	[38]
CH9	732	1315	∞	∞	36	478	527	5	-	[38]
CH10	653	1317	∞	∞	37	403	509	10	-	[38]
CH11	688	1328	∞	∞	38	401	494	11	-	[38]
CH12	713	1330	∞	∞	50	446	588	10	-	[38]
CH13	628	1329	∞	∞	38	369	450	9	-	[38]
CH14	642	1323	7	6	43	380	464	8	-	[38]
CH15	651	1303	∞	∞	49	402	561	14	-	[38]
CH16	627	1308	∞	7	32	384	445	8	-	[38]
CH17	579	1311	6	6	39	336	444	11	-	[38]
CH18	622	1338	∞	7	51	347	467	7	-	[38]
CH19	574	1333	7	7	48	329	445	14	-	[38]
CH20	499	1323	4	4	42	268	384	9	-	[38]
CH21	462	1330	∞	∞	30	253	332	16	-	[38]
CH22	474	1315	6	6	44	255	419	12	-	[38]

Switch Box Instances with inner pins										
name.cio	nets	col.	mcl.	d-mcl.	dens.	z.	d-z.	mp.	o.	ref.
mcc1	802	599	∞	∞	460	-	-	204	-	[38]
muxlps1	60	155	1	1	18	-	-	16	-	[38]
lcordic	11	320	2	2	6	-	-	2	+	[38]
lcord	11	320	2	2	6	-	-	2	+	[38]
lprim1	33	700	3	3	20	-	-	2	+	[38]
lprim1ss	33	85	3	3	20	-	-	2	+	[38]
rcord	10	320	2	2	7	-	-	2	+	[38]
rprim1	34	700	2	2	21	-	-	2	+	[38]

5.2 Channel visualization

We implemented CHAIRS[1] that creates a 3D image in tga format which shows the channel with all its pins and also the wires connecting them. Even obstacles can be illustrated. The image can be easily transformed into some commonly used format, e.g. jpg.

Since most of the mathematical computation that is necessary to create the image is done by the rendering software system POVRAY [39], the main task of CHAIRS is to provide an interface that transforms a CIO channel description into an appropriate POVRAY input file. Implementation of this interface was performed by using the script language PERL.

We included some useful features in CHAIRS. For example, it is possible to zoom out a part of the channel or to change the viewing position. This is helpful if the channel is huge and only a certain part of it is of interest. This interesting part may be located *behind* some other pins, wires or obstacles. That problem can be solved easily since the user is able to rotate the channel and/or to make obstacles transparent. Additionally, image size can be chosen by the user. Choosing the size as small as possible considerably speeds up the process of image construction on the screen.

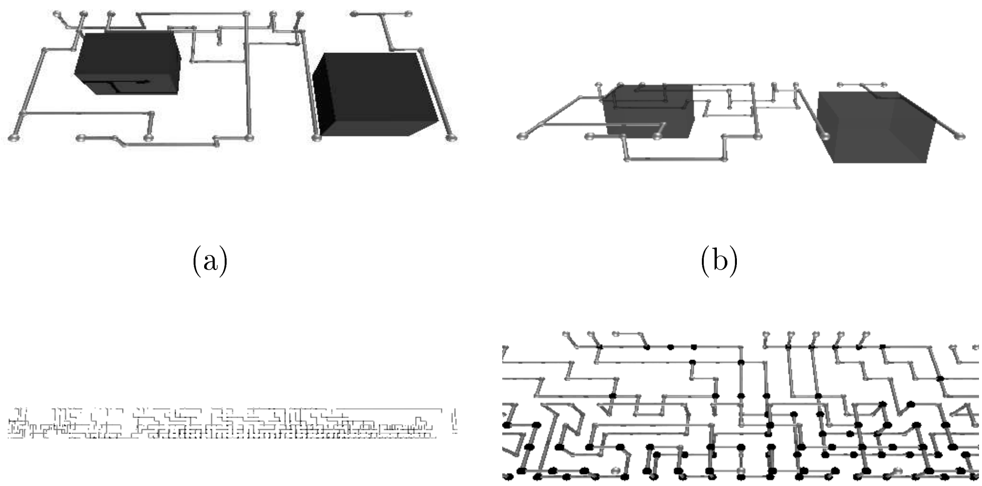
In the 3D images, pins are represented by balls of colour silver and golden cylinders are used in order to represent wires that realize connections between pins. If a channel contains multipoints (Section 2.3.8), there are black balls in the corresponding image at the locations of these multipoints. Finally, obstacles are represented by red boxes that are –dependent on the chosen parameter– more or less transparent.

Figure 5.1 shows some sample images generated by CHAIRS. In Figure 5.1 (a), there is a visualization of a small channel with 7 tracks, 14 columns, 2 layers and 4 nets. All the pins are connected and there is a non-transparent obstacle, too. In Figure 5.1 (b), the same channel is shown, but this time the obstacle has been made more transparent and the channel has been rotated. By this means, especially the wires that are not illustrated very well in (a) because of their location behind an obstacle or behind other wires come out better here.

An image of another channel is included in Figure 5.1 (c). Since the channel is big and shown completely in the figure, it is not possible to trace the wires that exist in the channel by eye. (d) shows a scaled up part of the same channel and thus the problem with the visualization in (c) can be avoided. (d) was generated by using -zoom and -dx parameters which like all other parameters are described in the following.

---

[1] **Cha**nnel **i**mage **r**endering **s**oftware



(a) (b)

(c) (d)

Figure 5.1: sample images generated by `CHAIRS`. (a) small channel, (b) same channel rotated with transparent obstacles, (c) big channel, (d) zoomed part of the channel in (c).

5.2.1 User manual

The command line must have the following format:

```
chairs [-h]|[-i <inputfile>][-o <outputfile>][-zoom <factor>]
       [-dx <offset>][-dy <offset>][-ob <transparency>]
       [-rot <degrees>] [-povW <width>] [-povH <height>]
```

Description of parameters:

`-h`: shows a short description of the software handling and does not perform any other steps.

`-i <inputfile>`: determines the filename of the file the channel data is read from. Default is `out.cio`.

`-o <outputfile>`: determines the filename of the file that is generated by `CHAIRS` and used as input file by `POVRAY`. Default is `out.pov`.

`-zoom <factor>`: scale up representation of the channel in the image by `<factor>`.

`-dx <offset>`: move channel for `<offset>` units in horizontal direction. One unit is exactly the distance between two pins that are located at two adjacent cells in the channel matrix.

`-dy <offset>`: move channel for `<offset>` units in vertical direction.

`-ob <transparency>`: include representation of existing obstacles into the 3D image. If `<transparency>` value is 0, nothing that is located behind an obstacle can be seen. The higher this value is, the more transparent is the representation of the obstacles and the better is the view on the wires behind the obstacles. If `<transparency>` is greater than 1, the user is not able to see obstacles any more because they are completely lucent in that case.

`-rot <degrees>`: rotate channel by `<degrees>` degrees around the horizontal axis.

`-povW <width>`: show image on screen in a window that is `<width>` pixels wide.

`-povH <height>`: show image on screen in a window that is `<height>` pixels high.

Bibliography

[1] Th. Hecker and M. Weber. Channlib – an advanced routing environment. Informatik-Bericht Nr. 28, Humbold-Universit"at zu Berlin, 1994.

[2] Christian Matuszewski. Datenstrukturen und Basisverdrahtungsstrategien für das Erlernen von Routingverfahren durch genetische Algorithmen. Master's thesis, Martin-Luther-Universtät Halle, August 1998. In german.

[3] Robby Schönfeld. Parametrisierte Maze-Router und Liniensuch-Algorithmen. Master's thesis, Martin-Luther-Universtät Halle, July 1998. In german.

[4] N. Göckel, R. Drechsler, and B. Becker. A multi-layer detailed routing approach based on evolutionary algorithms. In *Int'l Conference on Evolutionary Computation*, pages 557–562, 1997.

[5] F. Schmiedle, R. Drechsler, and B. Becker. Exact routing using symbolic representation. In *Int'l Symp. Circ. and Systems*, pages 394–397, 1999.

[6] T. Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout.* Teubner, Wiley, 1990.

[7] N. Sherwani, S. Bhingarde, and A. Panyam. *Routing in the Third Dimension - From VLSI Chips to MCMs.* IEEE Press, 1995.

[8] R. Kolla, P. Molitor, and H.G. Osthof. *Einführung in den VLSI - Entwurf.* B.G. Teubner, Stuttgart, 1989.

[9] A. Hashimoto and J. Stevens. Wire routing by optimizing channel assignment within large apertures. In *Proc. 8th IEEE Design Automation Workshop*, pages 155–169, 1971.

[10] David N. Deutsch. A "dogleg" channel router. In *Proc. 13th IEEE Design Automation Conference*, pages 425–433, 1976.

[11] Naveed Sherwani. *Algorithms for VLSI Physical Design Automation.* Kluwer Academic Publishers, Norwell, Massachusetts, second edition, 1995.

[12] Takeshi Yoshimura. An efficient channel router. In *Proc. 21th IEEE Design Automation Conference*, pages 38–44, 1984.

[13] B. W. Kernighan, D. G. Schweikert, and G. Persky. An optimum channel-routing algorithm for polycell layouts of integrated circuits. In *Proc. 10th IEEE Design Automation Conference*, pages 50–59, 1973.

[14] Takeshi Yoshimura and Ernest S. Kuh. Efficient algorithms for channel routing. *IEEE Transaction on Computer-Aided Design*, CAD-1(1):25–35, January 1982.

[15] K. Mikami and K. Tabuchi. A computer program for optimal routing of printed circuit connectors. *IFIPS Proceedings*, H47:1475–1478, 1968.

[16] D. W. Hightower. A solution to the line routing problem on a continious plane. In *Proc. 6th Design Automation Workshop*, 1969.

[17] R. .E .Tarjan. Data structures and network algorithms. In *CBMS-NFS Regional Conference Series in Applied Mathematics No. 44*, Philadelphia, PA, 1983.

[18] C. Y. Lee. An algorithm path connections and its applications. *IRE Transaction on Electronic Computers*, 10(3):346–365, 1961.

[19] R. K. Korn. An efficient variable-cost maze router. In *19th Design Automation Conference*, pages 425–431, 1982.

[20] N. Göckel, G. Pudelko, R. Drechsler, and B. Becker. A hybrid genetic algorithm for the channel routing problem. In *Int'l Symp. Circ. and Systems*, pages IV:675–IV:678, 1996.

[21] R. Drechsler. Verification of multi-valued logic networks. In *Int'l Symp. on Multi-Valued Logic*, pages 10–15, 1996.

[22] F. Somenzi. *CUDD: CU Decision Diagram Package Release 2.2.0.* University of Colorado at Boulder, 1998.

[23] A. Srinivasan, T. Kam, S. Malik, and R.E. Brayton. Algorithms for discrete function manipulation. In *Int'l Conf. on CAD*, pages 92–95, 1990.

[24] N. Göckel, R. Drechsler, and B. Becker. GAME: A software environment for using genetic algorithms in circuit design. In *Applications of Computer Systems*, 1997.

[25] R. Drechsler. *Evolutionary Algorithms for VLSI CAD.* Kluwer Academic Publisher, 1998.

[26] R. Drechsler, B. Becker, and N. Göckel. A genetic algorithm for variable ordering of OBDDs. *IEE Proceedings*, 143(6):364–368, 1996.

[27] R. Drechsler and N. Göckel. Minimization of BDDs by evolutionary algorithms. In *Int'l Workshop on Logic Synth.*, 1997.

[28] N. Drechsler and R. Drechsler. Exploiting don't cares during data sequencing using genetic algorithms. In *ASP Design Automation Conf.*, 1999.

[29] M. Keim, N. Drechsler, and B. Becker. Combining GAs and symbolic methods for high quality tests of sequential circuits. In *ASP Design Automation Conf.*, pages 315–318, Jan. 1999.

[30] J. Koza. *Genetic Programming - On the Programming of Computers by means of Natural Selection.* MIT Press, 1992.

[31] D.E. Goldberg. *Genetic Algorithms in Search, Optimization & Machine Learning.* Addision-Wesley Publisher Company, Inc., 1989.

[32] N. Drechsler, R. Drechsler, and B. Becker. A new model for multi-objective optimization in evolutionary algorithms. In *Int'l Conference on Computational Intelligence (Fuzzy Days)*, pages 108–117, 1999.

[33] Annett Thüring. Skriptbasierte Steuerung für parallele evolutionäre Algorithmen. Master's thesis, Martin-Luther-Universtät Halle, October 1999. In german.

[34] R. Drechsler and B. Becker. Learning heuristics by genetic algorithms. In *ASP Design Automation Conf.*, pages 349–352, 1995.

[35] R. Drechsler, N. Göckel, and B. Becker. Learning heuristics for OBDD minimization by evolutionary algorithms. In *Parallel Problem Solving from Nature, LNCS 1141*, pages 730–739, 1996.

[36] N. Göckel, R. Drechsler, and B. Becker. Learning heuristics for OKFDD minimization by evolutionary algorithms. In *ASP Design Automation Conf.*, pages 469–472, 1997.

[37] M. Burstein and R. Pelavinl. Hierarchical channel router. *INTEGRATION, the VLSI Jour.*, pages 21–38, 1983.

[38] DResearch Ines Peters. consultant.ines.peters@sietec.de, 1999.

[39] D. Wells and C. Young. *Ray Tracing Creations*. The Waite Group, 1993.