EINDHOVEN UNIVERSITY OF TECHNOLOGY

Eindhoven University of Technology

MASTER

Optimization of data path bit-slice placement

Nulens, M.P.M.L.

Award date:
1996

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

Department of Electrical Engineering

Eindhoven University of Technology

Design Automation Section (ES)

Optimization of Data Path Bit-slice Placement

By M.P.M.L. Nulens

Master Thesis

performed: April 1995 - February 1996
by order of prof.Dr.-Ing. J.A.G. Jess
supervised by ir. R.X.T. Nijssen

The Eindhoven University of Technology is not responsible for the contents of training and thesis reports.

Abstract

Due to recent advances in VLSI and ASIC designs technology, the size of chip designs is increasing, resulting in a larger chip area and a larger number of gates and nets per unit chip-area. The design specifications have become more strict, which makes it more difficult to generate layouts for these VLSI designs. This scale enlargement of the chips and the stricter design specifications make it necessary to develop new placement tools capable of chip area minimization without violating timing and layout constraints.

A commonly used layout method for VLSI designs is the row-based standard-cell placement, in which the circuits data path is divided into bit-slices that are placed in rows. In this thesis we describe the linear placement of data path bit-slices containing net-delay constraints and terminal constraints. The objective of the placement is to minimize the maximum net-density while not violating any net-delay constraints or terminal constraints.

After the modeling of the placement problem, a brief overview of different placement methods is given. Then two placements methods are worked out and finally the resulting linear placement tool is described.

The first proposed placement method is a force directed placement method. This placement method wasn't capable of doing correct net-delay constraint placement on critical paths.

Then a best first search method is used with success for the linear placement of bit-slices with net-delay constraints and terminal constraints. The resulting placement tool called HOPPER uses this best first search method to find a feasible placement. Several modification to the plain best first search algorithm are used to improve the quality of the placement, reduce the run-time, or limit the amount of used memory.

Contents

1 Introduction 1

2 Data-path placement 2
- 2.1 Placement . . . 2
- 2.2 Data-path . . . 3
- 2.3 The placement model . . . 4
- 2.4 The module model . . . 8
- 2.5 The constraints model . . . 9
- 2.6 Combinatorial Optimization . . . 12
- 2.7 An allowed placement . . . 13

3 Placement Heuristics 15
- 3.1 Exact versus approximation . . . 15
- 3.2 Some exact algorithms . . . 16
  - 3.2.1 Depth-first search . . . 17
  - 3.2.2 Breadth-first search . . . 17
  - 3.2.3 Best first search . . . 18
- 3.3 Some approximation algorithms . . . 18
  - 3.3.1 Successive augmentation . . . 19
  - 3.3.2 Force directed placement . . . 19
  - 3.3.3 Bipartitioning algorithms . . . 20
  - 3.3.4 Iterative improvement . . . 20
  - 3.3.5 Relaxation algorithm . . . 20
  - 3.3.6 Simulated annealing algorithm . . . 21
- 3.4 What algorithm to use ? . . . 22

4 Force Directed Placement 23
- 4.1 The model . . . 23
- 4.2 Module clustering . . . 24
  - 4.2.1 Dummy instances . . . 25
- 4.3 Net-Density calculation . . . 26
- 4.4 Shortcomings of the algorithm . . . 26
- 4.5 Conclusions . . . 27

5 **Best First Search** **28**
5.1 The algorithm . . . 28
5.2 Informedness of the A* search . . . 32
5.3 A monotonic heuristic function . . . 32

6 **HOPPER** **34**
6.1 Data flow . . . 34
6.2 Cost function calculation . . . 35
6.2.1 Monotonic cost function . . . 37
6.3 Modifications to the A* search . . . 38
6.4 The reduced successor set . . . 39
6.5 C-algorithm . . . 40
6.6 Epsilon algorithm . . . 40
6.7 The complex OPEN list . . . 42
6.8 Staged search . . . 42
6.9 Iterative density reduction . . . 43

7 **Related Topics** **46**
7.1 Circuit structure . . . 46
7.2 Articulation points . . . 47
7.3 Loops . . . 47

8 **Results** **49**
8.1 Circuits without constraints . . . 49
8.1.1 Plain A*-algorithm versus C-algorithm . . . 49
8.1.2 ϵ-algorithm . . . 50
8.1.3 OPENlist length reduction . . . 50
8.1.4 Iterative density reduction . . . 51
8.2 Circuits with constraints . . . 51
8.2.1 Plain A*-algorithm . . . 52
8.2.2 C-algorithm . . . 53
8.2.3 ϵ-algorithm . . . 53
8.2.4 Iterative density reduction . . . 53
8.2.5 OPENlist length reduction . . . 54
8.3 Summary . . . 54

9 **Conclusions** **56**
9.1 Future work . . . 57

References **58**

Bibliography **60**

A **Specification of constraints** **62**

B **Tables and figures** **64**

List of Tables

3.1 Analogy between physical systems and optimization problems. 21

B.1 Used test circuits 64
B.2 plain A*-search (optimum) versus C-algorithm 65
B.3 Results of the ϵ-algorithm 69
B.4 Results for OPEN-list length reduction 69
B.5 Results for iterative density reduction 70
B.6 Results for pure A*-search (optimum) with constraints 71
B.7 Results for C-algorithm with constraints 71
B.8 Results for ϵ-algorithm with constraints 72
B.9 Results for iterative density reduction with constraints 72
B.10 Results for OPEN-list length reduction with constraints 73
B.11 Results for OPEN-list length reduction with constraints (part-2) 74

List of Figures

2.1 Standard-cell placement . . . 3
2.2 Feed-through-cells . . . 4
2.3 Layout design-flow . . . 5
2.4 State-space graph G . . . 6
2.5 Search tree used for constrained linear placement. . . . 7
2.6 Bit-slice layout . . . 9
2.7 Terminal Cost function q() . . . 11
2.8 Net-length and constraint definitions . . . 12

3.1 Placement methods hierarchy . . . 16

4.1 Forces between the modules . . . 24
4.2 Module clustering . . . 25
4.3 Dummy modules . . . 26
4.4 Not feasible placement . . . 27

5.1 Equal depth front in BFS . . . 31
5.2 Cheapest cost front in Best-first search . . . 31
5.3 Influence of the heuristic function on the computational effort needed to conduct the search . . . 33

6.1 Sets used for incremental cost calculation . . . 35
6.2 Cost estimation for not placed modules . . . 37
6.3 Terminal constraint cost estimation due nets connected to not-placed modules 38
6.4 Terminal constraint cost due to nets connected to placed and not-placed modules 38
6.5 Terminal constraint cost due to nets connected to placed modules . . . 39
6.6 OPENlist structure . . . 43
6.7 Iterative density decrement control . . . 45

7.1 Schematic of a circuit with a pipeline structure . . . 46
7.2 Articulation points in an undirected graph . . . 47
7.3 A simple loop in a network . . . 48

B.1 OPENlist length reduction : OPENlist length versus placement cost . . . 65
B.2 OPENlist length reduction : OPENlist length versus number of visited nodes 66
B.3 OPENlist length reduction : OPENlist length versus number of visited nodes 66
B.4 OPENlist length reduction : OPENlist length versus number of visited nodestest 67
B.5 OPENlist length reduction : OPENlist length versus number of visited nodes 67

B.6 OPENlist length reduction : OPENlist length versus number of visited nodes 68
B.7 OPENlist length reduction : OPENlist length versus number of visited nodes 68

Chapter 1

Introduction

With the progress in VLSI and ASIC technology, designs have become too complex to design them by hand. Therefore design automation tools have become indispensable. Together with the increase of the size of the whole chip, the size of the modules in the integrated circuit is becoming smaller and the density of the modules on the chip is increasing.
The data in the VLSI-design is processed in a data path, which consists of a collection of modules connected by nets. Data-paths in VLSI circuits often contain very long paths and the design has to satisfy various design constraints. These can be constraints on the length of a critical path, or constraints on positions of terminals where external signals enter the data path. With the need of high-performance VLSI designs, operating at very high clock frequencies, the delays in module interconnections become very important. Together with the need for a short design time, *performance driven layout generation* is becoming more important.

In this thesis the placement of a data path bit-slice and some search algorithms used to find the placement will be described. The objective of the data path placement is to minimize the used area, taking into account several timing and layout constraints. This data path bit-slice placement is an essential part of a layout generation tool, which uses the circuit regularity to generate the layout.
Because many of the VLSI designs contain regular structures, we can try to use this information of regularity to guide the layout of the chip. Regular data paths often consist of parallel bit-slices which have the same structure. If we can extract the regularity of the circuit, and cluster it into bit-slices, than we only need to place one bit-slice. The rest of the bit-slices is then a copy of the placed bit-slice.
Due to the NP-completeness of the placement problem (see [Garey79]) we can save an enormous amount of compute time with this layout method, because the number of modules in one bit-slice is only a fraction of that in the entire placement. The saved time can be used to optimize the placement of that one bit-slice.
The two-dimensional placement problem is now converted into a repeated one-dimensional placements problem.

Chapter 2

Data-path placement

One of the last phases in the design process of a VLSI design is the layout synthesis. Since the number of modules in a design can be very large, we need efficient algorithms to layout the IC. The layout of the VLSI design should be as dense as possible to reduce capacitance of the wires, increase the speed of the data signals, reduce the used power and decrease the chip area.
With an increasing need for IC designs operating at very high clock frequencies, the placement of the modules has become more complex. The modules not only need to be placed efficiently on the IC area, but it's even more important to satisfy all the timing constraints resulting from the IC specifications.
Since most of the existing placement tools do not have the features to deal with specific constraints, it is my goal to develop a placement tool that can generate a placement, satisfying several timing and layout constraints, while still minimizing the used area.

2.1 Placement

The general placement problem is the problem of placing a number of modules, connected by a set of wires, on an area such that a certain cost function is minimized. Thus for each placement there is :

- A set of modules
- A set of wires interconnecting the modules
- A set of places on which the modules are to be paced
- A cost function as a measure of the quality of the placement

In modern VLSI design there are several types of placement. Each placement type has its own advantages and disadvantages. Three types of module placement over a two-dimensional chip area (see [Wong88]):

- Gate-array placement
- Standard-cell placement
- Macro/custom-cell placement

In this thesis we are interested in the *standard-cell placement.*
In the standard-cell design methodology, the *standard-cells* are defined in the library of a VLSI design package (such as COMPASS). The cells contain pre-defined logic functions. Their dimensions, performances, and electrical characteristics have already been determined. The cells, we call them modules, have an important property : all the modules have the same height, but modules of a different type can have a different width. We will use the most popular form of standard-cell placement, the row-based placement. In row-based placement modules are placed in rows, as shown in figure 2.1.

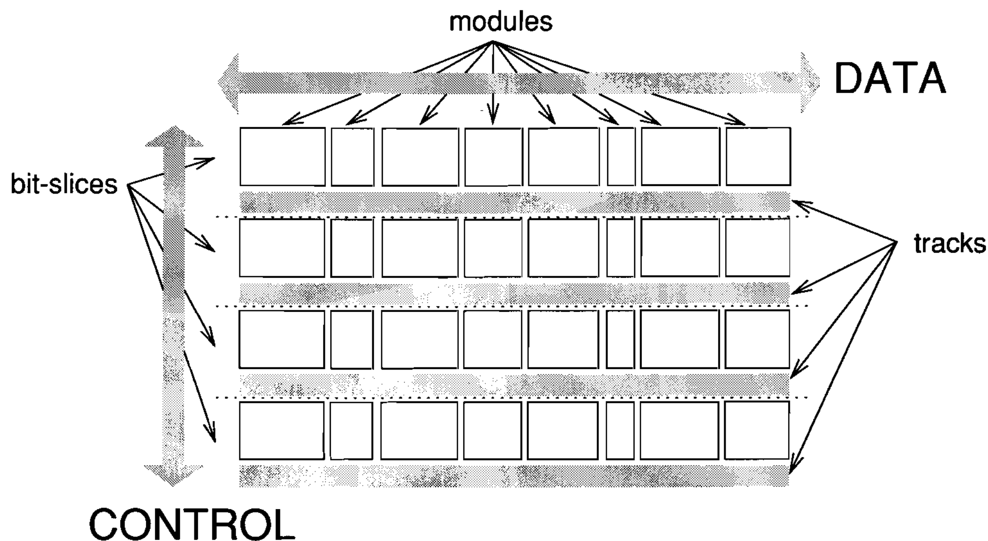

Figure 2.1: Standard-cell placement

Here are some definitions.

▷ **Definition 2.1.** *[Module]*
A standard-cell with a fixed height and a width that may differ from type to type.
□

▷ **Definition 2.2.** *[Placement]*
A placement is an assignment of set of modules, connected by a set of wires, to a set of places on a plane such that the modules do not overlap.
□

▷ **Definition 2.3.** *[Linear Placement]*
A linear placement is a placement with the set of places defined on a one-dimensional line.
□

2.2 Data-path

A data path in a VLSI design consists of a number of standard-cells connected by nets. The data path contains operators to perform the operations specified in the behavioral descrip-

tion and a control unit determines the sequence of the operations in the data path.
Since we use a row-based standard-cell placement, the components of the data path are layed out in a single row, i.e. in the form of a bit-slice. Rows are the bit-slices of the data path. Each bit-slice contains the modules necessary for the computation of one bit in the data path. See figure 2.1. Interconnections from one bit-slice to other bit-slices, or to the border of the IC is done through *feed-through-cells*. These are empty modules that are used to reserve enough space for the interconnections. Feedthrough cells are only needed if no layers are available for *over-the-cell* routing. See figure 2.2.

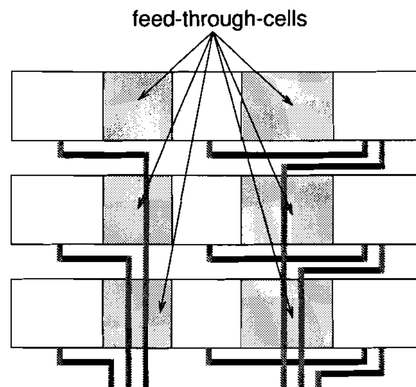

Figure 2.2: Feed-through-cells

The bit-sliced layout style is very suitable for the physical realization of these data paths because of its ability to minimize the area consumed by the interconnect. This is needed to produce a high performance circuit.

As we take a good look at a VLSI design, we can see that most of the large circuits have a certain amount of regularity. For the majority of the circuits the layout of the data path will be nearly optimal if a bit-slice layout is applied to them. One of the most important properties of regular data paths is that the placement of each bit-slice is alike. All the slices have the same placement and modules in the same column have the same width. If the placement of one bit-slice is known, the placement of the entire data path is known. With this knowledge the design-flow of the layout looks like in figure 2.3.

2.3 The placement model

We use the mathematical model as used in [Cederbaum74].

The *set of modules* to place $B = \{b_1, b_2, \ldots, b_n\}$, with n the number of modules to place.

The modules in B must be placed in some arbitrary *module order* $D = (b_{k_1}, b_{k_2}, \ldots, b_{k_n})$, and $b_{k_i} \in B$, with k_i an invertable mapping from $\{1, \ldots, n\}$ onto $\{1, \ldots, n\}$.

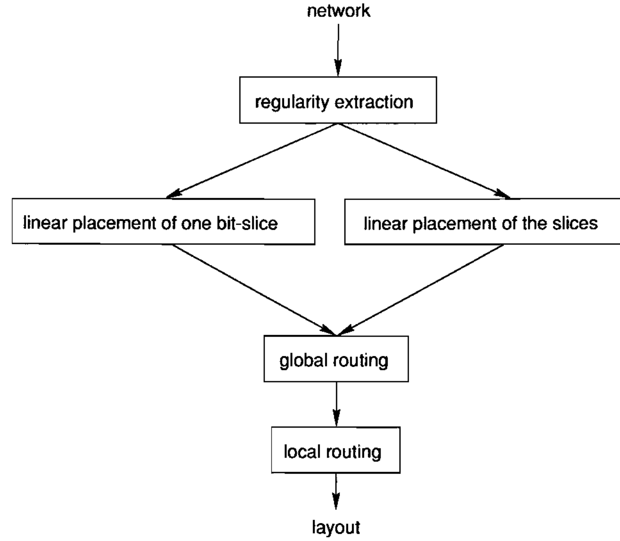

Figure 2.3: Layout design-flow

Let A be a subset of B, $A \subseteq B$. Then $N(A)$ is the *set of nets* connected to at least one of the modules in A. The *net-density* $d(A)$ is the number of nets that connect the modules belonging to A with the remaining modules in $B - A$.

▷ **Definition 2.4.** *[Net-density]*
The net-density $d(A)$ is the number of interconnections between the set of modules in A and the set of modules in $B - A$. Another notion used is d_i.

$$d(A) = |N(A) \cap N(B - A)| \quad (2.1)$$

$$d_i = \text{net-density after module } b_{k_i}, \text{ in a placement with module order}(b_{k_1}, b_{k_2}, \ldots, b_{k_i}, b_{k_{i+1}}, \ldots, b_{k_n}), \text{ with } 1 \leq i < n \quad (2.2)$$

□

The modules are placed in a row. The order in which the modules are placed determines the net density and the length of the nets which connect the modules. We want the net-length of certain nets as short as possible to minimize the data delay time, and to satisfy the time constraints. In an indirect way the module order influences the the *data path delay.*
The goal of our placement tool is to minimize the net-density and to satisfy all the constraints. Because of the correlation between the net-density, the total wire-length and the used chip area, a minimization of the net-density will also result in minimization of the used chip area.

The collection of placements, and partial placements in a placement problem can represented as *search space*. This *search space* can be represented as an acyclic graph $G(V, E)$, where each vertex in V is a subset of B and E is the set of edges. Each edge defines a transition from a partial placement D_k into D_i. With $D_i = D_k + \{b_j\}$ and $b_j \notin D_k$

In figure 2.4 the graph is shown.

On level k there are $\binom{n}{k}$ subsets of B. The *level* of a node we mean the number of nodes in the path to that node.
The total number of vertices in the graph is :

$$\sum_{k=0}^{n} \binom{n}{k} \tag{2.3}$$

Equation 2.3 can be simplified by a special case of Newtons binomium :

$$\begin{aligned} \sum_{k=0}^{n} \binom{n}{k} &= \sum_{k=0}^{n} \binom{n}{k} a^k b^{n-k}, \text{with } a = 1, b = 1 \\ &= \sum_{k=0}^{n} \binom{n}{k} 1^k 1^{n-k} \\ &= (1+1)^n \\ &= 2^n \end{aligned} \tag{2.4}$$

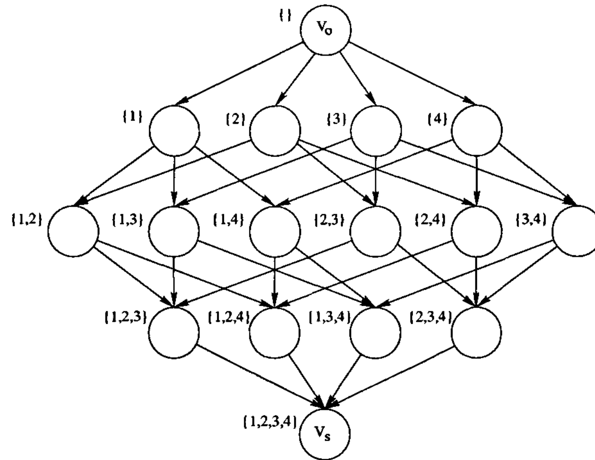

Figure 2.4: State-space graph G

The source of the graph is vertex v_0, which corresponds with the empty set ϕ. From v_0 leave n edges to the vertices $v_i \sim \{b_i\}$, $i = 1, 2, \ldots, n$.
The graph G has a sink $v_s \sim B$, with n incoming edges.
Thus G is a regular graph of degree n.
For each vertex in G the sum of the in- and out-going edges is n. The overall number of edges in G is :

$$\frac{n \cdot 2^n}{2} = n \cdot 2^{n-1} \tag{2.5}$$

Each directed path from v_0 to v_s is cycle free, contains n edges and every vertex on that path is visited once. The overall number of paths from v_0 to v_s is $n!$

Each source-to-sink path in G is a one-to-one correspondence with a module order. For example the path :

$$Q_k = (\phi, \{b_{k_1}\}, \{b_{k_1}, b_{k_2}\}, \ldots, \{b_{k_1}, b_{k_2}, \ldots, b_{k_{n-1}}\}, B)$$

corresponds with the module order $O_k = (b_{k_1}, b_{k_2}, \ldots, b_{k_n})$

The argument of an edge from vertex $v_i \sim B_i$ to vertex $v_j \sim B_j$ is the net-density $d(B_i)$. All the edges leaving an edge v_i have the same argument.

This placement model is very useful for placements where each module ordering is a valid placement. These are placements where only net-density or wire-length minimization is concerned. Since we have to deal with constraints such that not every module ordering is a valid placement, we use a search tree as showed in figure 2.5 in which each node can hold the path cost and the number of a module. This search tree is a *permutation tree*. This has the advantage that the cost of a node can *not* be overwritten by a the cost of the cheaper path to that node. This is necessary since it is possible that the cheaper path will eventually result in a not-allowed placement, but the forgotten path, would lead to an allowed placement. This would throw away allowed placements or just the only allowed placement.

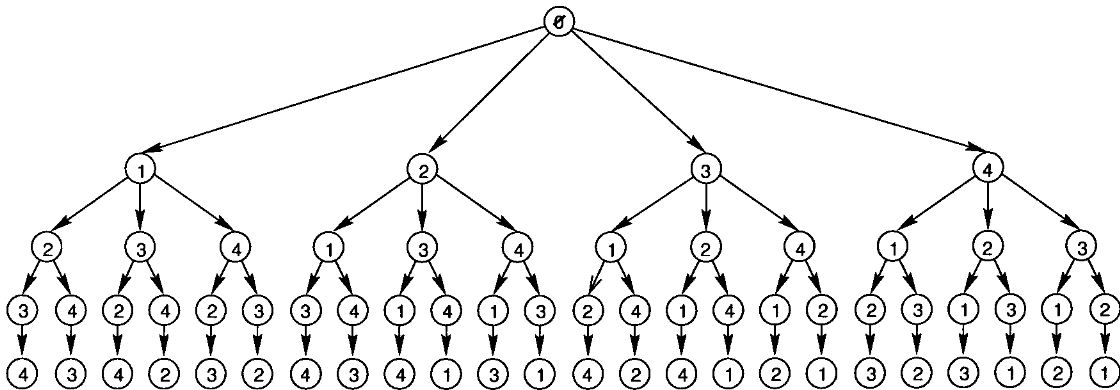

Figure 2.5: Search tree used for constrained linear placement.

The number of paths in this tree is equal to the number of paths in the state graph of figure 2.4, this is $n!$.

In the search tree at level k there are $\frac{n!}{(n-k)!}$ nodes.

The total number of nodes in the search tree is :

$$\sum_{k=0}^{n} \frac{n!}{(n-k)!} = \sum_{k=0}^{n} \frac{n!}{k!} \tag{2.6}$$

$$\text{with } e \doteq \sum_{k=0}^{n} \frac{1}{k!} \quad , \quad \text{for } n \text{ large enough}$$

$$\sum_{k=0}^{n} \frac{n!}{(n-k)!} \doteq n! \cdot e \quad (2.7)$$

The number of edges in the search tree is :

$$\sum_{k=0}^{n} \frac{n!}{(n-k)!} \cdot (n-k) = \sum_{k=0}^{n} \frac{n!}{k!} \cdot k \quad (2.8)$$

The number of nodes in the search tree is enormous for even small numbers of n. If we would have to enumerate the whole search space to find an allowed placement it would almost take forever even for relative small n. For example, for $n = 20$ there are, according to formula 2.7, $6.613 \cdot 10^{18}$ states in the search space. If it would be possible to enumerate 10.000 nodes per second, than it would take over 20.000.000 years to find a solution. This example shows that the enumeration of the whole search space is not possible, even for a small number of modules.
Formula 2.7 shows that the number of states in the search tree increases dramatically with the number of modules in the placement. This shows the need for more advanced search and/or placement methods to avoid enumeration of the whole search space.

2.4 The module model

The modules that we are going to use are all standard-cells. All the cells have the same height, that is the height of one row, so they can be placed side-by-side in a one-dimensional row. Terminals are available at the top or the bottom side of the module. The standard-cells are made in CMOS technology [Eijndhoven92].
The modules have the following properties :

- rectangular shape
- fixed height
- variable width
- connectors at the top and/or the bottom side

In figure 2.6 an example of a module layout is shown. We use no over-the-cell routing, all the wires are placed in a *routing channel*. If it would be necessary to use over-the-cell routing, the model can easily be accommodated to it. Over-the-cell routing would only decrease the number of tracks in the routing channel.

▷ **Definition 2.5.** *[Track]*
A narrow area parallel to a modules in a bit-slice, reserved for the routing of nets.

□

▷ **Definition 2.6.** *[Routing channel]*
Place used for tracks required for data signal wires outside the module.

□

The number of tracks needed to route all the wires can be larger (but never smaller) than the maximum net density. This *track density* will be known after the local routing has been done.

▷ **Definition 2.7.** *[Track-density]*

The number of tracks in the routing channel.

$$\text{track-density} \geq \text{net-density} \quad (2.9)$$

□

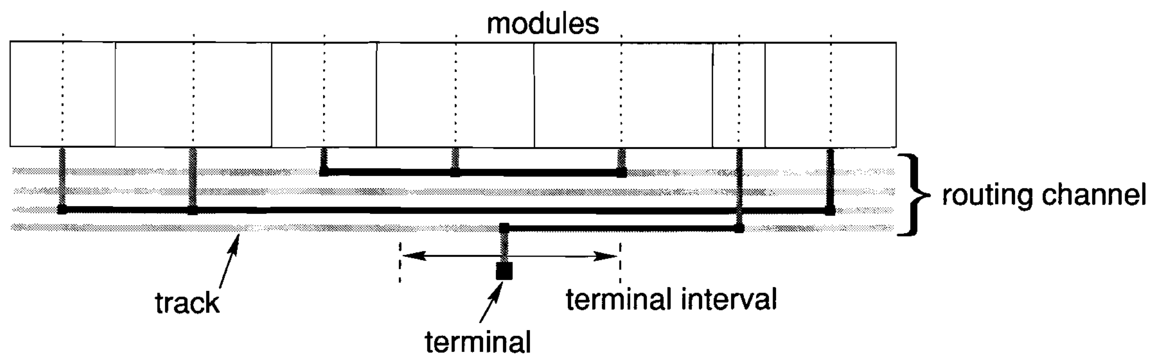

Figure 2.6: Bit-slice layout

2.5 The constraints model

The use of a cost function in combination with constraints introduces two different constraint models. These constraint models are :

1. If a constraint is violated then we assign very high cost to the (partial-)placement in which this constraint is exceeded. But we keep the (partial-)placement in the search tree. This gives the placement program to ability to allow certain constraint violations. This can be useful if the specified constraints are too strict and no feasible placement exist. We can then search for the placement with the least constraint violations.

2. If a constraint is violated then the (partial-)placement in which this constraint is violated is immediately removed from the search tree. Here we don't allow any constraints to be exceeded.

The developed linear placement tool can work with both of these constraint models. Which constraint model is used can be chosen by a command line option.
The second constraint model can be seen of a special case of the first constraint model in which violated constraint are assigned a cost value ∞. In the next chapters we use the first constraint model in which (partial-)placement with constraint violations are assigned very high cost values.

To generate a placement in conformation to its specification, it is often necessary to restrict the placement, and define net-delay constraints on some nets in the critical path. If these net-delay constraints are violated, the whole placement is not feasible. If the data path block will be used within a larger design, it is often necessary to specify the places where the data path block communicates with the rest of the chip. This is necessary to minimize net-length, and decrease net-delay. The external connections in the data path block are called *terminals*.

▷ **Definition 2.8.** *[Terminal]*
External connector of the data path block.

□

The places where the terminals enter the data path block can be predefined. Modules connected to these terminals must be placed within the specified *terminal-interval.*

▷ **Definition 2.9.** *[Terminal-interval]*
An interval (a, b), that defines the place where a module can make a connection to a terminal. With a and b positions along the bottom of the bit-slice, relative to to the most bottom-left position of the bit-slice. The dimension a and b is [λm].

□

Each terminal is connected to only one net. If a terminal is connected to more than one net that this is an illegal situation, that should not occur!

To satisfy the timing constraints it must be possible to specify the length of critical nets. This to ensure that data on critical nets arrives in time.
The constraint types are :

- **Terminal constraints** : The designer can specify the place of the terminal connection. The place is given as a terminal-interval. See figure 2.6.

- **Net delay constraints** : To satisfy specific timing of a critical net, a certain maximal net-length may not be exceeded. In practice we are interested in path-delay. This path-delay has to be converted into net-length constraints of the nets in the path.

Of course it is possible to combine constraints. These are nets with a net-constraint, that are connected to a terminal with a terminal-constraint. This ensures that certain modules are placed within a certain range from a terminal.

▷ **Definition 2.10.** *[Terminal constraint]*
An instance of a terminal constraint is formalized as a pair $[T, x]$, where T is the name of the terminal and x is the associated terminal interval. The cost of a net j connected to a terminal T with a terminal constraint are :

$$t(T, x_j) = \begin{cases} q(\min(e-a, e-d)) & \text{if } d \leq e \\ q(\min(e-a, c-e)) & \text{if } e \leq c \\ 0 & \text{if } c \leq e \leq d \end{cases} \tag{2.10}$$

$$\text{with } x = [a, b] \text{ the terminal interval of } T \tag{2.11}$$

$$e = \frac{b-a}{2},\ j \text{ is a net with interval } [c, d]$$

$$\text{and } q(y) = k \cdot y^2, \text{ with } q(e) = k_{unity} \tag{2.12}$$

□

The function $q(y)$ assigns to each terminal position y a cost. Figure 2.7 shows the behaviour of $q()$. In this picture k_{unity} is an arbitrary value for the cost of a the terminal constraint at the border of the terminal interval.

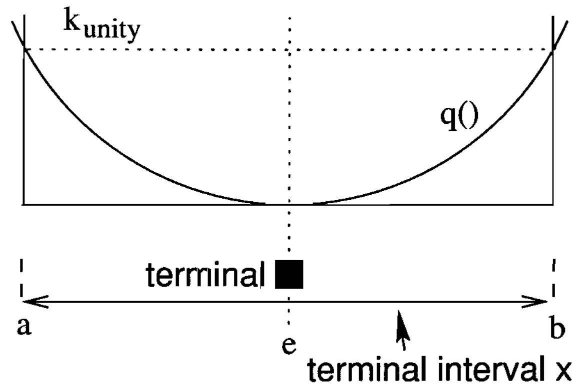

Figure 2.7: Terminal Cost function q()

▷ **Definition 2.11.** *[Net delay constraint]*
An instance of a net delay constraint is formalized as a pair (j, l_{max}), where j is the name of the net and l_{max} the maximum length of this net. The cost of a net j is :

$$r(j) = \begin{cases} 0 & \text{if } length(j) \leq l_{max} \\ length(j) - l_{max} & \text{if } length(j) > l_{max} \end{cases} \tag{2.13}$$

□

To compare the quality of different placements of the same circuit, a cost function is used. The cost function has the form :

$$f(i) = k_1 \cdot d_{max} + k_2 \cdot C_{net\,delay\,constraints} + k_3 \cdot C_{terminal\,constraints} \tag{2.14}$$

$$d_{max} := \max(0 \leq i < n : d_i) \quad \text{, maximum net-density} \tag{2.15}$$

$$C_{net\,delay\,constraints} := \sum_{i \in N(B)} r(i) \tag{2.16}$$

$$C_{terminal\,constraints} := \sum_{j \in N(B)} \sum_{m \in \text{set of terminals}} t(m,j) \tag{2.17}$$

$$\tag{2.18}$$

In figure 2.8 the net-length is displayed as l_1 and l_2. The distance between the middle of the terminal interval and the terminal position is displayed as y_1. y_1 is used a parameter for $q()$. Since we don't have any information about the position where wires enter a module, it is assumed that all the wires are connected to the middle of the module.
The constants k_1, k_2 and k_3 are used to balance the cost function. Changing these *balance* factors can be used to guide the placement algorithm, since changing these parameters will change the definition of a good placement.

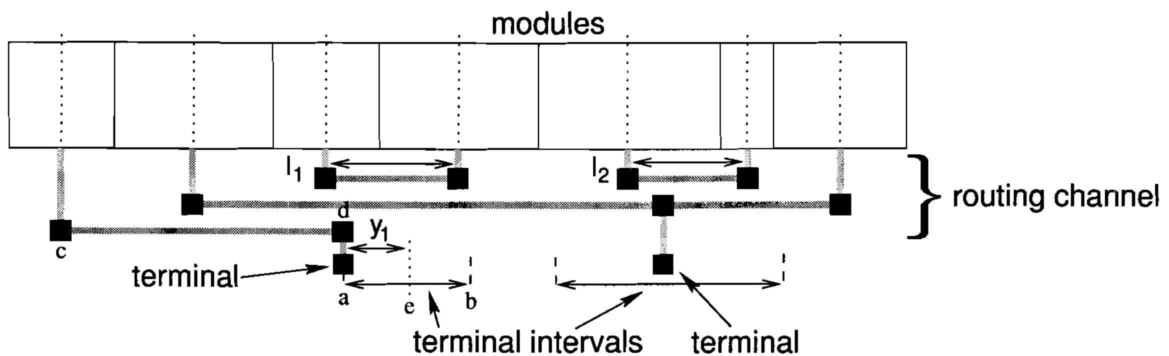

Figure 2.8: Net-length and constraint definitions

2.6 Combinatorial Optimization

Large numbers of practical problems belong to the class of combinatorial optimization problems. The placement problem is also one of them.
The objective of a combinatorial optimization problem is to find a "best" solution out of the *solution space*. A possible solution that satisfies the set of constraints is called a *configuration*. To discriminate between the different configurations we use the *cost function* $f(x)$ to a configuration x. To find a good configuration we only need to minimize the cost function.
Some well known problems such as the *Traveling Salesman Problem* (see [Cormen92]) are combinatorial problems.
Unfortunately optimum linear placement minimizing the maximum net-density has been proved to be NP-complete (see [Garey79]). Due to the NP-completeness of the placement problem we know that there won't be a linear placement algorithm that finds the optimum solution within polynomial time.

Thus we know that we can't find the optimum solution in polynomial time. But if we can't find an optimum solution we can try to find a near optimum solution.

▷ **Definition 2.12.** *[Combinatorial Optimization Problem]*
An *instance* of a combinatorial optimization problem is formalized as a pair (S, f) where the *finite* or *countably infinite* solution space S is defined as:

$$S = \{ \text{ all possible solutions } \},$$

and $f : S \rightarrow R$ is a cost function over S. The *objective* is to find an optimal solution $i_{opt} \in S$ such that

$$f(i_{opt}) \leq f(i) \text{ forall } i \in S.$$

□

Since we are only interested in the one-dimensional placement, our placement problem is an instance of the combinatorial optimization problem, and known to be NP-complete [Karp72].

▷ **Definition 2.13.** *[Linear Placement Problem (LPP)]*
Let $M = \{M_i, \ldots, M_m\}$ be a set of *modules* and $N = \{N_i, \ldots, N_n\}$ a set of nets, where $N_j \in P(M)$ for $1 \leq j \leq n$. Then the *linear placement problem* is specified by the pair (S, f) with $S \in$ *solution space* and f a value of the *cost function*,

$$S = \{\text{all possible linear placements}\},$$

and f is a *cost function*, defined such that

$$\forall\, p \in S : f(p) = \text{ an estimate of the quality of the placement}$$

□

2.7 An allowed placement

With the definitions made in the previous sections we can decide which placements are allowed and which aren't.

▷ **Definition 2.14.** *[Allowed placement]*
A module placement is allowed if:

- modules are placed at legitimate locations without overlapping
- there is enough space to implement interconnections
- net delay constraints are not violated
- terminal constraints are not violated

  What is meant with the violation of constraint depends on the constraint model. If the constraints model uses high cost values for constraint violations then a certain bound is used to define how much constraint violation is allowed.

□

With this definition of an allowed placement we can define the optimal placement.

▷ **Definition 2.15.** *[Optimal placement]*
An allowed placement $a \in S$ is an optimal placement if,

$$\forall_{b \in S} : f(a) \leq f(b), \quad S \text{ is the set of allowed placements.}$$

□

Since we know that the LPP is an NP-hard problem, we know that the search for the *optimal solution* is not always possible, especially for bigger problem instances. This implies that that the LLP is *intractable*.

Chapter 3

Placement Heuristics

The placement problem is a complex problem in which the quality of the placement depends on several factors. Since there isn't an analytic way to calculate the optimum or near optimum solution, we prefer to use heuristics to search for a feasible solution.
All of the heuristics described in the following subsections have a different way of searching the state space of the placement. One of the main difficulties in searching the state space is the discontinuity of the cost function of the placement problem. Due to the constraints in the placement the cost function of the placement can be very discontinuous. Allowed placements with a low cost function can lay beside not allowed placements that have high cost functions due to constraint violations. This often result in a dead-end search path. Another difficulty in searching the state space is the existence of many *local optima* in the state space, This makes the search for an optimal or near optimal solution difficult since algorithms can be trapped in a local optimum.
There exist many different placement heuristics, a few of them are described in the following paragraphs. In figure 3.1 the relations between the different placement heuristics are shown. The figure makes separates two main categories of heuristics, these are the heuristics that search for the optimum solution and the heuristics that search for a near optimum solution.

▷ **Definition 3.16.** *[Local optimum]*
A graph $G(V,E)$ has a local optimum in $c \in V$, if a surrounding $U(c)$ of c exists, in which $f(c) \leq f(x)$ for all $x \in U(c) \cap V$, and $f(x)$ the cost of node x.

□

3.1 Exact versus approximation

The placement problem is an intractable optimization problem. For an optimization problem one may try to find an optimal or near optimal solution by and *exact* algorithm or *approximation* algorithm (see [Cormen92]) respectively. An exact algorithm for an intractable problem has worst case super polynomial time complexity. Consequently the use of exact algorithms becomes impracticable when dealing with larger problem instances. Therefore one usually resorts to using approximation algorithms when dealing with larger instances of intractable optimization problems.

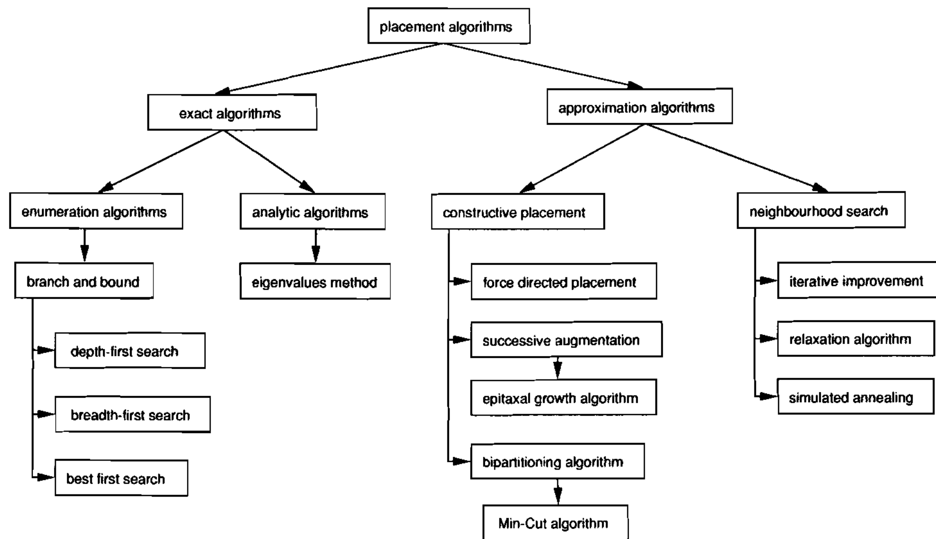

Figure 3.1: Placement methods hierarchy

If a search strategy always finds the cheapest path to the goal then the search strategy is called *optimal* (see [Shinghal92]).

▹ **Definition 3.17.** *[Optimality of a search strategy]*
A search strategy is said to be optimal if, and only if, it is guaranteed that the final solution is also the cheapest solution.

□

Besides the quality of a solution, it is important to know if the used search strategy finds a solution, if one exists. A search strategy can be classified as being either *complete* or *incomplete* (see [Shinghal92]).

▹ **Definition 3.18.** *[Completeness of a search strategy]*
A search strategy is said to be complete if, and only if, it is guaranteed to find a solution path provided such a path exists.

□

Thus an ideal placement heuristic should be *complete* and *optimal*. But due to the NP-hardness of the the placement problem, the optimality of a heuristic implies that the algorithm doesn't find a solution within polynomial time. Therefore an optimal algorithm can only be used on relative small or special cases of intractable problem instances.

3.2 Some exact algorithms

One class of exact algorithms for the placement problem is that of the *enumeration* algorithms. Enumeration algorithms find an optimum by searching the state space S, evaluating and comparing costs of encountered states.
Within the class of the enumeration algorithms there are algorithms based on the *branch and*

bound method (see [Shinghal92]), such as the best first search. These algorithms search the state space visiting states that lay within certain limits. These limits are the upper and lower values of the cost function.
All the search algorithms have the following ingredients :

- an *initial state;*
- a *goal test* on states detecting if a state is a solution;
- a set of operations that can be applied to change the current state;
- a cost function to indicate the quality of a state.

Examples of these *enumeration* algorithms are : *depth-first search, breadth-first search* and *best-first search.*

3.2.1 Depth-first search

The DFS method (see [Cormen92]) starts at the *source* node, at depth 0. With the *depth* of node we mean the number of nodes on the path to that node.
At each node it enters an unvisited edge if this edge leads to a partial solution that satisfies all the constraints. This algorithm thus first goes deep into the search tree. If the visited node is not valid the algorithm *backtracks* until it finds a node with a unvisited edge. This edge is then entered. and a new node is visited. We repeat this search process until the solution is found, or there are no nodes left to visit.
Properties:

- complete, if backtracking is used;
- optimal if backtracking is used;
- uses very little memory.

3.2.2 Breadth-first search

The BFS method (see [Cormen92]) starts at the *source* node, which is at depth 0. This node is expanded, generating all its successors which now become nodes at depth 1. After all the nodes of depth d have been generated, the nodes of depth $d+1$ are generated, where $d \geq 0$. If there are nodes with more than one incoming edge, the value of the node will be relaxed to the cheapest path to the node. The algorithm ends if all the nodes of level n have been generated. This simple BFS algorithm needs a complete enumeration of the entire search space to find the solution.
Since the number of nodes at each depth may grow exponentially, *combinatorial explosion*, the BFS will need serious amounts of memory.
Properties:

- complete;
- optimal;
- needs huge amounts of memory.

3.2.3 Best first search

The best-first search (see [Shinghal92],[Cai90],[Nakao93]) starts with expanding the *source* node. The new generated nodes, that haven't been expanded yet, are placed on an OPEN-list. From then on, at any given time, we select that node for expansion which is the cheapest, that is the most promising, among all the existing nodes in the OPEN-list. The selected node is expanded and the same process is repeated, until a solution is found.
The main difference with the DFS or the BFS algorithms is the form of the cost function. Let:

$g(x)$ be the cost of the cheapest path from the source node to node x and $g^*(x)$ is the real cost from the source node to node x, where $0 \leq g^*(x) \leq g(x)$;

$h(x)$ be the estimation of the cost of the cheapest path from node x to the destination node, where $0 \leq h(x) \leq h^*(x)$. Important : $h(x)$ may not be an over-estimation of the real cost $h^*(x)$ of the cheapest path from node x to the destination node, to ensure the optimality of the search.; and

$f(x) = g(x) + h(x)$.

Therefore, $f(x)$ is the estimate of cost of the cheapest solution path that is constrained to pass through x. Here $h(x)$ gives the algorithm the information of which path to choose.
Algorithms derived from the best first search are : *A^* search, Greedy search, Beam Search and Uniform Cost search.*
Properties:

- complete;
- optimal, if $0 \leq h(x) \leq h^*(x)$;
- needs large amounts of memory to keep all the unexpanded nodes in memory.

3.3 Some approximation algorithms

In the class of approximation algorithms we distinguish *constructive placement* and placement by *neighborhood search* algorithms.
Constructive placement algorithms build up placements in a similar way humans would do. These algorithms try to place the modules by looking at the modules and their interconnections, and then decide where the modules have to be placed. In general the quality of the solution is intermediate and the computation time is little. But their major disadvantage is that they are very problem specific. Examples of constructive algorithms are : *successive augmentation (see [Sutanthavibul93]), force directed placement (see [Jeske94]), and bipartitioning algorithms (see [Kerninghan70],[Lauther79]).*

neighborhood search algorithms start from an initial placement that is improved iteratively by applying changes to a temporary placement. The initial placement doesn't have to a feasible solution, it just needs to contain the the whole set of modules. The algorithm tries to improve the solution, or to make it an *allowed placement*.
If the changes made to the placement each iteration are not too big, the cost-function can be calculated incrementally. This can drastically decrease the computation time, but mostly,

this is more complicated to implement.
The weakness of these *iterative improvement* methods is that they easily get stuck in a local minimum, and the quality of the placement strongly depends on the starting solution.
The *iterative improvement* method is an algorithm applicable in many different types of problems.
Examples of neighborhood search algorithms are : *relaxation algorithm (see [Mathur94]), iterative improvement algorithm (see [Doll94]), simulated annealing algorithm (see [Wong88],[Sechen87]).*

3.3.1 Successive augmentation

A placement method that works well when there are a lot of constraints is the *successive augmentation* approach. This algorithm tries to place one module at a time, without violating the constraints. An example of method is given in [Sutanthavibul93]. In this article an *adaptive algorithm* is presented to satisfy all the constraints and optimize the solution.
Another example of an successive augmentation method is the *epitaxial growth algorithm* (see [Schweikert76]).
The *epitaxial growth algorithm* starts by placing one or more important or special modules, having many connections to other modules. Next the algorithm searches for an unplaced module having the maximum number of connections to the placed modules. This module is placed at the best possible position, taking into account the cost-function and the constraints. This process continues until no unplaced modules are left to be placed.
Properties of successive augmentation algorithms:

- not complete;
- not optimal;
- fast algorithm;
- moderate results, since the quality of the placement strongly depends on the "intelligence" of the algorithm.

3.3.2 Force directed placement

In force directed placement modules are regarded as small objects, the nets are regarded as elastic bands. The stretching coefficient of the bands is adjustable, this allows the bands to be either very stretchable or total static. With this mechanism we allow modules to be very mobile or be very static.
The elastic bands pull the modules towards each other, with a force proportional to the distance between these modules.
To avoid module overlap the modules edges produce a repulsive force, which affects all the other modules. The closer two modules are placed together, the more the two modules repulse each other. A placement solution is now a stable state of the model.
The implementation of constraints is also possible in this model. Terminal constraints can be regarded as some kind of magnet that contracts the participated modules. Net constraints can be implemented by varying the elastic coefficient of the elastic bands.
Properties :

- not complete;

- not optimal;
- moderate quality of the placement, since the number of found solutions is limited;
- fast algorithm.

3.3.3 Bipartitioning algorithms

Bipartitioning algorithms are based on the *divide and conquer* strategy. These algorithms divide the group of modules into two disjunct subsets. Here the number of net interconnections between the sets is minimized. This method is based on the finding the minimum cut through a graph in such a way that the total size of all modules in each set is about equal and the number of connections between the two sets is minimized. This algorithm is then used recursively on the subgroups, until the remaining subsets contain one module.
To find a reasonable solution it is necessary that the size of the sets is balanced. If no set size balancing is done a trivial solution will be found with on one side the empty set and the other side a set with all the modules in the placement.
Since the number of modules to look at is drastically reduced, this kind of algorithms can find a feasible solution fast. The well known *Min-Cut algorithm* (see [Lauther79]) is an example of a bipartitioning algorithm.
Properties:

- not complete;
- not optimal;
- fast algorithm.

3.3.4 Iterative improvement

Starting from an initial placement, The iterative improvement algorithm (see [Doll94]) changes a small part of a placement. If the change was for the better then the change is accepted, and a new iteration is started.
This algorithm easily gets stuck in a local optimum since it only accepts changes that lower the cost function. Since the cost function of the placement with constraints contains many local optima, this algorithm is not applicable for our problem.

- not complete;
- not optimal;
- may get stuck in local minima.

3.3.5 Relaxation algorithm

The relaxation algorithm (see [Mathur94]) starts with an arbitrary module, and searches for the best position. If this position was already occupied by another module then the module is still put on that position. The module that occupied the place is moved to another another best place. This process is repeated until the best best position is unoccupied, or a module that has already been moved would move again. After a number of found sub-solutions

have been found, a new relaxation is started from the sub-solution with the cheapest cost function.

- not complete;
- not optimal;
- may take long to find a feasible placement.

3.3.6 Simulated annealing algorithm

An extension to the iterative improvement algorithm is the *simulated annealing algorithm* (see [Wong88],[Sechen78]). To overcome the problem of getting stuck in a local optima the simulated annealing accepts changes that lead to an increase of the costs (*Hill-Climbing*), but with a probability that is slowly decreased to 0. The probability satisfies the Boltzmann's distribution,

$$e^{E(r)/k_bT} \tag{3.1}$$

Where $E(r)$ is the energy associated with state r, and k_b is the Boltzmann's constant.
This process works similar to the annealing of a solid material. A solid material is annealed by raising its temperature to a maximum value at which the material is in a liquid phase, in which the particles are randomly arranged. Then the temperature is lowered very slowly so the solid can reach a low-energy state. This low-energy state is equal to growing a single large crystal, in an optimization problem this is a near optimal solution. The simulating annealing algorithm is even optimal if the annealing scheme is very slowly, $t \downarrow 0$.
We now can see a analogy between physical systems and optimization problems. In figure 3.1 the analogies are displayed.

Table 3.1: Analogy between physical systems and optimization problems.

Physical Systems	Optimization Problems
State	Solution
Energy	Cost-Function
Ground State	Optimal Solution
Rapid Quenching	Iterative Improvement
Careful Annealing	Simulated Annealing

Properties:

- complete;
- not optimal, unless annealing scheme is very slowly, $t \downarrow 0$;
- general applicable algorithm;
- quality of the result is a function of the run-time.

3.4 What algorithm to use ?

We want the placement algorithm to have certain parameters which have a predictable influence according to:

- memory usage
- quality of the placement
- run-time

Since our placement problem has to deal with some constraints, we could choose for the *force directed placement* or the *successive augmentation method*. These algorithms are fast, but do not produce high quality placements. The *best-first search algorithm* is very promising in search for a good solution and it's cost function is very flexible for use with constraints. The disadvantage of the best-first search is the huge amounts of memory needed for even small problem instances.
A property of the net-delay constraints is the adjacent placement of the modules connected to the same net with a net-delay constraint. This property could be used in the force directed placement.
Terminal constraints can be very irregular, this asks for a very flexible placement heuristic. The constraints introduce many local optima in the placement search space.
In the next chapter we will try to develop a placement algorithm based on force-directed placement. This method is chosen for its speed and it's ability to handle constraints.
In the following chapter a placement algorithm based on the best-first search is described. The best first search doesn't get stuck in a local optimum very easily, it has a flexible cost-function, which is necessary to implement the constraints, and it is general applicable.

Chapter 4

Force Directed Placement

The placement heuristic explained in this chapter is based on a *force directed placement (FDP)*. At first sight this method seemed very promising in finding a good placement in relatively short time. In the end though it appeared that the heuristic couldn't deal satisfactorily with net-constraints. Nevertheless it brought some good insight in the placement problem.

4.1 The model

The basic idea behind the *force directed placement* is the following. In the beginning of the placement all modules lay on a pile. The nets, interconnecting the modules can be seen as elastic bands. The elasticity coefficient of the can be adjusted, to be sure that some modules are connected more strong than others. To introduce a downward force, assume that all modules have a certain weight.
If we take one module, lifting it up, other modules will follow. When no more modules are on the ground, a partial placement has been made. The distance between the modules is a measure for the connectivity between the modules. The stretched network introduces some kind of clustering, that can be used as the first phase in a placement heuristic.

The **force directed placement** uses a physical model to place the modules. Several forces are defined on the modules. The quality of the placement is proportional to the amount of tension between the modules. Minimization of the forces will lead to a minimization of the net-density and net-length.
There are three different forces involved. See figure 4.1. These are :

1. **Net forces**. These are the forces that apply to the module due to the nets, which are modeled as elastic bands. These forces minimize the net-length and the net-density.

2. **Module edge forces**. These are forces that apply to the edges of the modules. These are necessary to prevent module overlapping.

3. **Terminal forces**. These forces apply to modules attached to terminals.

The implementation of constraints can be done by using bands with a higher elasticity coefficient, or even a stiff connection.

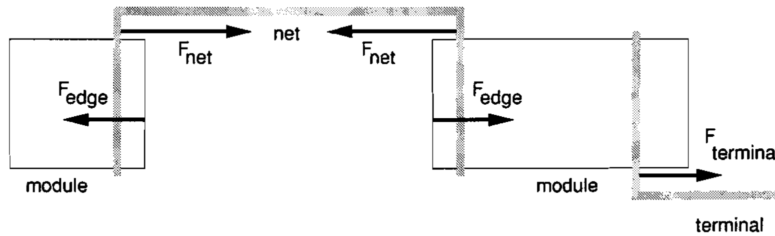

Figure 4.1: Forces between the modules

The forces applied to each module satisfy the following equation:

$$F_i(s) = \sum_{\forall j \in B} F_{egde}(i,j) + \sum_{\forall k \in N(i)} F_{net}(k) + \sum_{\forall l \in \text{Terminals}} F_{terminal}(l) \quad (4.1)$$

$$= 0 \quad (4.2)$$

with

$$F_{edge}(i,j) = \text{edge force from module } j \text{ on module } i \quad (4.3)$$

$$F_{net}(k) = \text{net force due to net } k \quad (4.4)$$

$$F_{terminal} = \text{force on module due to terminal } l \quad (4.5)$$

We want that for each module the sum of the forces applied to the module equals zero. For each module there is an equation in of the form of equation 4.2. This system of non-linear equation can be solved by an *Newton Raphson* iteration process (see [Eijndhoven92]). This would result in an unique absolute position s for each module in the linear placement.
In the force directed placement each selection of the first module to place introduces only one placement. If there are n modules in the placement then the force directed placement can only generate n different placements.

4.2 Module clustering

To get the exact forces and distances in the network, we could use a brute force method to solve a system of force equations. This method would be too expensive in terms of run-time. Therefore a clustering method could be used to divide the total placement into smaller local placement problems.

The clustering can be done by dividing the set of modules into sets of modules that should be placed somewhere near the modules in a previous or next set. Thus modules that have a high connectivity should be placed adjacent to each other. Figure 4.2 shows an example a such a module clustering. The clustering method is in fact a *breadth-first* way of walking through the network graph.

The steps in the clustering algorithm is :

1. Take a module, the source s, and place it at the left side of the row. This is cluster C_0

2. Get the successors of the previous cluster. This will be cluster C_i. The successors are all the modules connected the modules in cluster C_{i-1}, and that are not contained in a previous cluster.

3. If there are still modules not in a cluster, continue at (2) with C_{i+1}

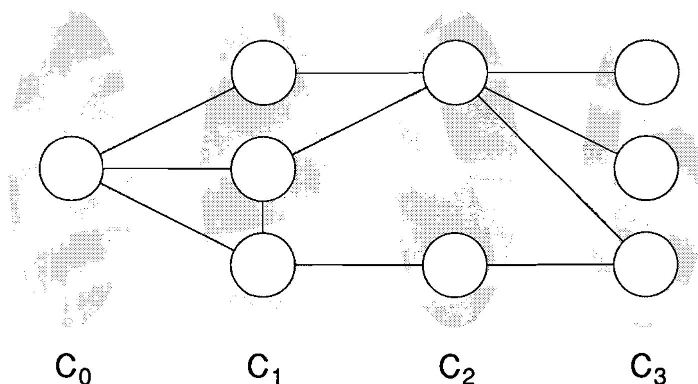

Figure 4.2: Module clustering

The number of the cluster is the same as the *ASAP* index, starting at the source module s.

This clustering can be done very quickly. Each modules is visited only one to build up this cluster. Which gives it time complexity $\mathcal{O}(n)$.
With this module clustering, connected modules will be placed, in the same cluster, or in an adjacent cluster. This brings up an important *invariant* for this clustering :
In this clustering each cluster C_i has only nets to next cluster C_{i+1} or to the previous cluster C_{i-1}.

If we now place each cluster separately, than the total placement is the concatenation of the placements of the clusters. Thus each cluster has to be placed separately.

4.2.1 Dummy instances

The terminal constraints introduce a problem with the module clustering invariant. Terminals with constraints have fixed positions (within the terminal interval). This introduces the possibility for a net to cross one of more clusters, which would violate the clustering invariant.
To hold the clustering invariant *dummy instances* are introduced. This are modules without width. They are only used to hold the clustering invariant. Figure 4.3 shown an example of such a dummy modules.

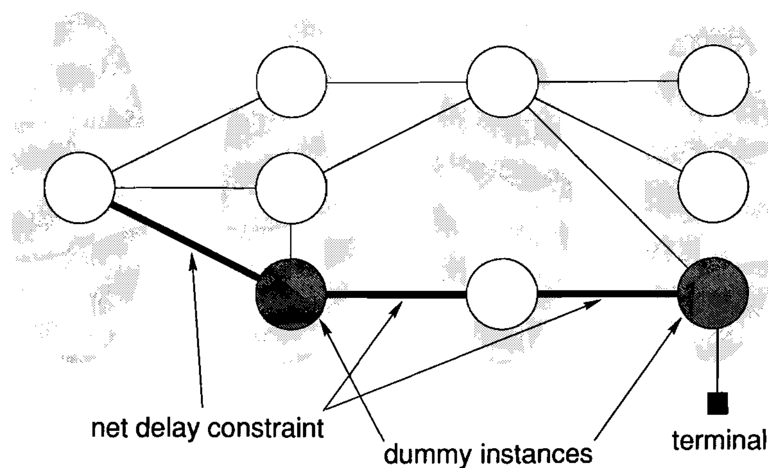

Figure 4.3: Dummy modules

4.3 Net-Density calculation

One of the great advantages of the clustering is the simple net-density calculation. In the clustering modules of cluster C_i can have connections to the previous cluster C_{i-1} or to next cluster C_{i+1}. With this property the net-density between two clusters can be calculated as :

$$d(C_i, C_{i+1}) = |N(C_i) \bigcap N(C_{i+1})| \tag{4.6}$$

Since the number of modules in a cluster C_j isn't very large for most of the placements, the evaluation of equation 4.6 is much faster than the evaluation of equation 2.1.
The net-density calculation with the clusters as also much faster since the number of modules and nets if much smaller.

4.4 Shortcomings of the algorithm

The effectiveness of the clustering depends on the selection of the first modules to place, and the connectivity of the circuit. For fully connected circuits the clustering would lead to two clusters; one with the first placed module, the other cluster with the rest of the modules. This would leave use with almost the same big cluster we started with. For circuits with a large number of interconnection the clustering would lead to very large clusters, making the effect of the clustering very small.

The use of terminal constraints and net-delay constraint introduces some major difficulties. The first difficulty can be seen in figure 4.3. The introduction of dummy modules makes the the total number of modules to place larger and increases the time to find a placement.

Another difficulty for the clustering a *critical paths*. These critical paths consist of a number of nets with net-delay constraints. If the critical path lays across several clusters, then it

is possible that the clustering makes it impossible to find a feasible placement. A cluster can be too wide, making it impossible to satisfy a certain net-constraint. Figure 4.4 shows an example of this situation, where the clustering disables the ability to find a feasible placement.

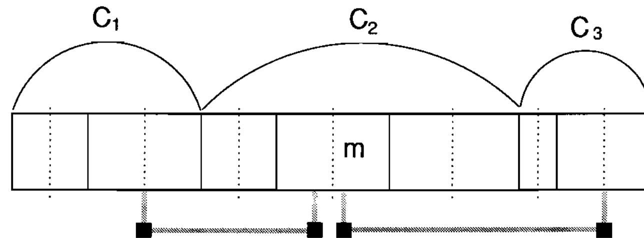

Figure 4.4: Not feasible placement

4.5 Conclusions

The clustering method as proposed for the force directed placement cannot be used in combination with net-delay constraints. Which makes this method of force directed placement not usable for the linear placement problem with net-delay constraints. It can however be used for net-density minimization in circuits without net-delay constraints.

Chapter 5

Best First Search

A search method that is very popular in the artificial intelligence is the *best-first search* (see [Shinghal92]). A small description has been given in section 3.2.3.
Since the Force Directed Placement failed on the implementation of the constraints in the placement, a best-first search is used to search for an allowed placement. The best first search is a generally applicable algorithm, that doesn't need the human like "intuition" of the force directed placement.

5.1 The algorithm

The best-first search has been a framework for heuristics which speed up algorithms by using semantic information about a domain. A*-search is a special case of the best-first search algorithms. It uses an estimator function $f(u, d)$ to estimate the cost of the shortest path between node u and the destination node d.
The algorithm starts with expanding the *source* node. With *expanding* node i we mean calculating all the successors of node i. The new generated nodes, that haven't been expanded yet, are placed on an OPEN-list. From then on, at any given time, we select that node for expansion which is the cheapest, that is the most promising, among all the existing nodes in the OPEN-list. The selected node is removed from the OPEN-list, put on the CLOSED-list and is then expanded. The same process is repeated, until a solution is found.
The OPEN-list contains all nodes, that have been generated, but not yet expanded. The cost of the nodes is used to sort the OPEN-list.
The CLOSED-list is only used to detect cycles in the search process and relax the cost of nodes that are already in the CLOSED-list.
A* has been quite influential due to its optimality properties. A best-first search without estimator functions is not very different from Dijkstra's shortest path algorithm (see [Cormen92]) when applied to the linear arrangement. The time complexity of Dijkstra's shortest path algorithm is $\mathcal{O}(|V|^2)$.

The main difference with the DFS or the BFS algorithms is the form of the cost function. Let:

$g^*(x)$ be the cost of the cheapest path from the source node to node x, where $g^*(x) \geq 0$;

$h^*(x)$ be the cost of the cheapest path from node x to the destination node, where $h^*(x) \geq 0$; and

$f^*(x) = g^*(x) + h^*(x)$.

This $f^*(x)$ is the exact value of the shortest path through node x. This exact value of $f^*(x)$ is not known during the search. Thus an estimation $f(x)$ of $f^*(x)$ is used instead. Let:

$g(x)$ be the estimation of the cost of the cheapest path from the source node to node x, where $0 \leq g^*(x) \leq g(x)$. In most of the search problems, and also in the linear placement problem, this $g(x)$ is exactly known.;

$h(x)$ be the estimation of the cheapest path from node x to the destination node, where $0 \leq h(x) \leq h^*(x)$; and

$f(x) = g(x) + h(x)$. This is the used cost function.

Here $h(x)$ gives the algorithm the information about things to come.

There are two special cases of the A*-algorithm :

- $h(x) = 0, \rightarrow f(x) = g(x)$, There is no information about the path from x to the solution. This search is called *uniform cost* search or *incurred cost* search. This search method is very similar to the Dijkstra's shortest path algorithm.
- $g(x) = 0, \rightarrow f(x) = h(x)$, The information from the path to node x is left out, we only look at the estimation of the path to come. The search is also known as the *greedy* search.

Assuming that the graph does not contain negative edge costs and that the cost function f is monotonic increasing with the depth in the search tree. And if the heuristic function h never overestimates the cost of the shortest path (u, d), then as soon the destination node d, with $h(d, d) = 0$, is selected and removed from the OPEN-list, its path (s, d) contains the cheapest path from source node s to d.

One of the best properties of the best-first search algorithm, is that *in most cases* it does not have to examine all the nodes to discover the cheapest path to the destination. Furthermore, the estimator can provide extra information to focus the search on the shortest path to the destination, reducing the number of nodes to be examined.

The algorithm :

```
State* AStar(State : InitalState) {
  State : i, n;
  SortedList : OPEN, CLOSED;
  Set : Successors;

  OPEN.add(InitialState);

  while ( OPEN.length() > 0 ) {
    n = OPEN.front(); // select the next node to expand and remove if from OPEN
    CLOSED.add(n);
```

```
      if ( IsSolution(n) )
         return &n; // return solution
      Successors = CalcSuccessors(n);
      foreach (i in Successors) {
         i.predecessor = &n; // set pointer to predecessor
         i.cost = CalcCost(i);
         if ( !CLOSED.contains(i) && !OPEN.contains(i) ) {
            OPEN.add(i);
         } else if ( CLOSED.contains(i) ) {
            old = CLOSED.find(i);
            if ( old.cost > i.cost ) {
               CLOSED.remove(old);
               old = i;
               OPEN.add(old);
            }
         } else if ( OPEN.contains(i) ) {
            old = OPEN.find(i);
            if ( old.cost > i.cost ) {
               OPEN.remove(old);
               old = i;
               OPEN.add(old);
            }
         } else
            OPEN.add(i);
      }
   }
   return NULL;
}
```

The search tree used for the linear placement problem is acyclic. Each state is different and can only be expanded once. Thus if a state is moved from OPEN to CLOSED, then it can't appear on OPEN again. Therefore CLOSED is now superfluous. This will speed up the search process since we don't have to search for states on OPEN of CLOSED. The reduced A*-algorithm used for linear placement :

```
State* AStar(State :  InitalState) {
   State :  i, n;
   SortedList :  OPEN;
   Set :  Successors;

   OPEN.add(InitialState);

   while ( OPEN.length() > 0 ) {
      n = OPEN.front(); // select the next node to expand and remove if from OPEN
      if ( IsSolution(n) )
         return &n; // return solution
```

```
        Successors = CalcSuccessors(n);
        foreach (i in Successors)
            i.predecessor = &n; // set pointer to predecessor
            i.cost = CalcCost(i);
            OPEN.add(i);
    }
    return NULL;
}
```

Another important difference with the BFS method is the form of the expanded node front. The breadth-first search method characterizing the Dijkstra algorithm proceeds along the contours of equal depth, the best-first search though proceeds along the contours of cheapest cost. See figures 5.1 and 5.2. This is an advantage since at all times the cheapest nodes are available to continue the search with. The best-first search can jump through the state space always continuing with the cheapest node. This property ensures that we cannot get stuck in a local optimum, if the OPEN-list is long enough to accommodate the entire front.
In the placement problem the equal cost contour is a disadvantage. The OPEN list will contain states of different path-length. The states are sorted on their path-cost, although it is not quite correct to compare states (partial placements) of different levels. In the linear placement ordering, states from different depth in the search tree represent partial placements with a different number of modules placed. These states can have the same cost although the number of placed modules is different.

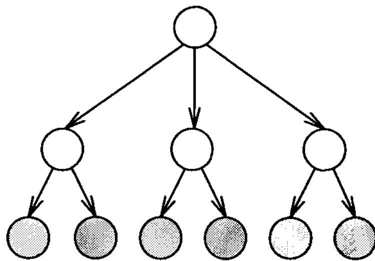

Figure 5.1: Equal depth front in BFS

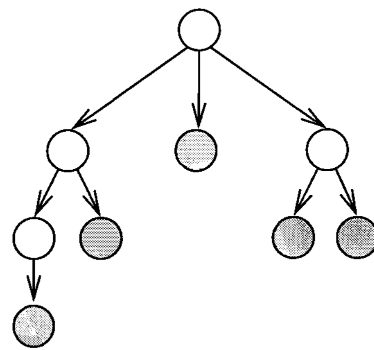

Figure 5.2: Cheapest cost front in Best-first search

What is the time complexity of the A*-algorithm applied to the linear placement problem ? In the worst case the whole permutation tree has to bee enumerated. There are $n! \cdot e$ (eq. 2.7) nodes (partial placements) in the search tree. Each of these nodes has to be removed from sorted OPEN-list. The OPEN-list is implemented as a *red-black tree* (see [Cormen92]). This step takes $\mathcal{O}(\log n)$ since we assume that the length of the open list is $k \cdot n$, with k a constant. The calculation of the the successors takes $0.5 \cdot n$, assuming that each node has and average of $0.5 \cdot n$ successors. For each successor its cost are calculated, this is done in constant time c. And finally the each successor is put on the OPEN-list which takes $\mathcal{O}(\log n)$ time.
The generation of the successors of a node takes now : $0.5cdotn(c + \log n)$. Al together the

time complexity is :

$$\mathcal{O}(n! \cdot e(0.5 \cdot n(c + \log n) + \log n)) \doteq \mathcal{O}(n! \cdot \log n) \tag{5.1}$$

This time-complexity looks very bad. In practical problems however the algorithm needs only to visit a small fraction of the nodes in the search space. The time needed to find a placement strongly depends on the connectivity of the circuit. The worst case is a fully connected circuit.

5.2 Informedness of the A* search

A big difference of the A* search compared to the DFS and the BFS is that the A* search is an *informed* strategy. In general search strategies are *informed* or *uninformed*. In the literature the uninformed strategies are also called *blind, weak* or *general-purpose* strategies. Strategies that are informed employ some problem specific knowledge in search of a solution. This makes the informed strategy a bit "smarter" than the uninformed strategy, which will result in better solutions and/or less states to evaluate until the solution is found.
The A*-algorithm is an informed strategy due to the use of the estimation $h(x)$ of the cheapest path from node x to the destination node. The smaller the difference between the estimation $h(x)$ and the real cost $h^*(x)$ to the destination, the less states need to be visited to find a near optimum or even the optimum solution.

5.3 A monotonic heuristic function

In order to ensure that the first found solution is also the cheapest one, the used cost function $f(x)$ must be a *monotonic* increasing with length of the search path. If $f(x)$ would not be monotonic then the cost of a node of level i could be lower than the cost of a node of level j, with $j < i$. Then it would not be possible to ensure than the cheapest solution was found as soon as a node of level n is selected from the OPEN-list. A not monotonic cost function $f(x)$ would make it necessary to enumerate the whole search space to find the best solution.

▷ **Definition 5.19.** *[monotonic]*
We call $f(x)$ monotonic if, for all nodes x and y, where x is a node on the path from source to node y.
$f(x) \leq f(y)$

□

The cost function has to be monotonically increasing with the length of the path. This is necessary to ensure that the first found solution is also the cheapest one. This can be seen from :

1. If the cost function is monotonically increasing then : $f(p) \leq f(c_p)$, with p a node in the graph and c_p all the nodes reachable from node p.

2. The invariant of the OPENlist :
   $f(\text{OPENlist.front()}) \leq f(y), \forall y$ in OPENlist

3. If a node x is the selected for expanding then it is selected from the OPENlist. Its successors are generated and put on OPENlist, and node x is removed from the OPENlist.

These three points imply that at all times during the the search holds that :

$$f(\text{OPENlist.front()}) \leq f(g), \text{with } g \text{ all nodes reachable from all nodes on OPENlist.} \quad (5.2)$$

If now OPENList.front() is a solution then it is also the cheapest solution.

If $f^*(x)$ is cost of optimal solution then :

- A* will expand all nodes for which $f(n) < f(x)$
- A* will expand some nodes for which $f(n) = f(x)$
- A* will not expand nodes for which $f(n) > f(x)$

For the A* to construct the smallest possible search tree such that no node is expanded more than once, it is necessary that :

1. the heuristic function $h(x)$ is monotonic; and
2. $h(x)$ is the highest possible lower bound on $h^*(x)$.

In the optimal situation we want $h(x) = h^*(x)$. This would imply that we need all the information and a lot of computational effort to calculate the highest upper-bound for $h(x)$. This would decrease the number of nodes to visit, but cost a lot of extra time to calculate the exact value of $h(x)$. The other extreme would be $h(x) = 0$ (uniform cost search), this is the lowest under-bound for $h(x)$. Now the number of states to visit with be the highest, but the is no time needed to calculate $h(x)$.
From this we can see that is a tradeoff between the time to calculate $h(x)$ and the number of nodes to visit. The fastest search is somewhere in the middle. This is displayed in figure 5.3

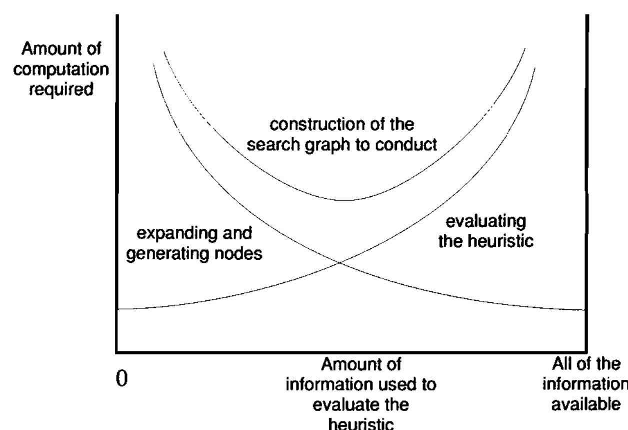

Figure 5.3: Influence of the heuristic function on the computational effort needed to conduct the search

Chapter 6

HOPPER

The theory from the previous chapters has resulted in the linear placement tool called HOPPER. It is based on the best-first search algorithm (see [Cai90],[Nakao93]), but has several other algorithms and options in it to guide and speedup the search process. The name HOPPER comes from the property of best-first search to jump from one cheapest node to the next cheapest node in the cheapest cost front.

HOPPER reads the circuit information from a net-list file. The placement results are saved in an order-file and in a BACKDRAW-file, used to visualize the placement.

The main search algorithm used in HOPPER is the A*-algorithm as displayed in the previous chapter. Several modifications have been made to the basic algorithm. With some command-line options the search behaves like :

1. pure A*-search : best used for placements with a lot constraints;

2. Uniform cost search : Used for placements without constraints, net-density minimization;

3. C-algorithm : A Fast Constructive DFS method used to calculate an upper bound on the net-density;

4. ϵ-algorithm : Approximation algorithm based on A*-algorithm.

In the following sections the differences between these search methods will be explained.

6.1 Data flow

HOPPER was designed to generate linear placements, containing constraints, from net-list files. It can be used in combination with VLSI layout packages such as COMPASS. The layout done by compass is often not very satisfying, external linear placement is required. The data-flow could be like :

1. COMPASS - Logic Assistant

2. COMPASS - Chip Compiler

3. Save floor-plan in COMPASS format
4. Save floor-plan in EDIF format (flattened)
5. Convert Edif-format net-list to Netlist format (Edif2NL)
6. Generate Linear ordering with HOPPER
7. Convert order-file to COMPASS format (not yet available)
8. Merge order-file with COMPASS floor-plan (not yet available)
9. Import floor-plan in COMPASS - Chip Compiler

6.2 Cost function calculation

As we have seen in the previous chapter, it is very important to find a good cost function that doesn't need too much time to calculate. Since the A*-search jumps from one local optimum to the next one, a pure incremental cost calculation is not possible. It would cost too much memory to store all the needed information.
But certain parts of the cost function need only be calculated once for all the successors of a state. Figure 6.1 shows the sets of nets and modules used to calculate the cost function.

The placement is build up from left to right.

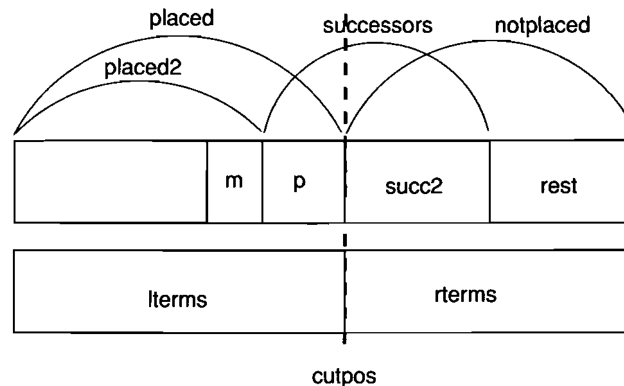

Figure 6.1: Sets used for incremental cost calculation

If a node m is selected for expanding, all the successors of this node are calculated.
PLACED2 is the set of placed modules inclusive the last placed module that is contained in m. The SUCCESSORS set contains all the successors of m. The REST set contains the modules not in (PLACED2 $\cup$ SUCCESSORS). The sets PLACED2, SUCCESSORS and REST are constant for all the successors of m. These sets are constant for all the successors of m are calculated once.
The rest of the sets needs to be calculated for each successor p of m. The different parts of the cost function are :

Net-density:

$$\begin{aligned} d(\text{PLACED}) &= |(N(\text{PLACED2}) \cup N(p) \cup N(\text{LTERMS})) \cap \\ &\quad (N(\text{NOTPLACED}) \cup N(\text{RTERMS}))| \quad (6.1) \\ \text{LTERMS} &= \text{set of terminals to the left of CUTPOS} \\ \text{RTERMS} &= \text{set of terminals to the right of CUTPOS} \end{aligned}$$

Constraint cost:

$$\begin{aligned} C_{net\,delay\,constraints} &= C_{net}(\text{PLACED2}) + C_{net}(p) + C_{net}(\text{NOTPLACED}) \quad (6.2) \\ \text{with } C_{net}(A) &= \text{net delay cost due to modules in } A \\ C_{terminal\,constraints} &= C_{term}(\text{PLACED2}) + C_{term}(p) + C_{term}(\text{NOTPLACED}) \quad (6.3) \\ \text{with } C_{term}(A) &= \text{terminal constraint cost due to modules in } A \\ & \quad (6.4) \end{aligned}$$

The cost function is :

$$\begin{aligned} f(p) &= g(p) + h(p), \text{ with} \\ g(p) &= k_1 \cdot max(d(\text{PLACED2}), d(\text{PLACED})) + \\ &\quad k_2 \cdot C_{net}(\text{PLACED2}) + C_{net}(p) + \\ &\quad k_3 \cdot C_{term}(\text{PLACED2}) + C_{term}(p) \quad (6.5) \\ h(p) &= k_2 \cdot C_{net}(\text{NOTPLACED}) + \\ &\quad k_3 \cdot C_{term}(\text{NOTPLACED}) \quad (6.6) \end{aligned}$$

The balancing factors k_1, k_2 *and* k_3 are necessary to allow that certain costs are more important than others. If we would make k_1 very large compared to k_2 and k_3 than the net-density has become the most important factor in the cost function. These balancing factors can be used to set a priority on what we would like to minimize in the placement, the net-density, net-delay constraint, the terminal constraints, or a combination of these.

The heuristic function $h(p)$ is the forecast for the constraints cost due to the not yet placed modules. In figure 6.2 is displayed what is considered in the constraint cost forecast. In the figure a indicates an estimation of the net-length from a net from the set of placed modules to the set of not-placed modules. In the same figure b indicates the forecast on the terminal cost and the net-delay cost for a net from a module in the not-places set to a terminal to the left of the cut-position.
There is no estimation for the net-density. It is very difficult to give a good net-density estimation for the path to come. It depends on the connectivity and the structure of the circuit which can be very different. In order to find the cheapest solution the net-density may not be over-estimated. Therefore we use no net-density estimation. Thus if no constraints are involved, the cost function is $f(p) = g(p)$, the algorithm uses an uniform cost search for net-density minimization.
As soon as constraints are involved HOPPER will use an A* search and achieve the best results.

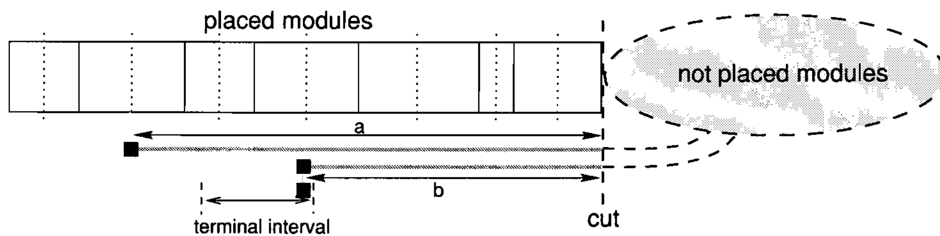

Figure 6.2: Cost estimation for not placed modules

6.2.1 Monotonic cost function

As stated before, the cost function has to be monotonically increasing to ensure that the cheapest solution is also first one that will be found.
To ensure that the cost function is monotonically increasing the different parts of the cost function are calculated as describe below. We use the sets as shown in figure 6.1

net-density

$$\text{net density cost} = max(d(\text{PLACED2}), d(\text{PLACED})) \tag{6.7}$$

This is monotonically increasing since we take the maximum net-density of the previous and the net-density just after the last placement module.

net-delay constraints

The placement starts with no modules placed and ends with all modules placed. During the placement modules are added to the most left available place. The cut-position, that is the position after the last placed module, travels from left to right during the placement.
The net-delay constraints for the nets that interconnect the modules in the PLACED can only increase during the placement. The net-delay constraint cost are monotonically increasing with the net length, see definition 2.11. The set of placed modules grows, and so does the total physical width of the set of placed modules. We calculate only that part of the net that is known, this is the part of the nets that lay to the left of the cut-position. Figure 6.2 display this situation. From the above follows that the net-delay constraint cost can only increase as the partial placement contains more placed modules.

terminal constraints

The model of the terminal constraints is defined a parabolic cost function on the terminal interval. This makes it possible for the terminal constraint cost (see figure 2.7) to decrease or increase when a terminal position is moved within the terminal interval. The cost function could decrease if during the placement the terminal is moved towards the center of the terminal interval. This behaviour would conflict with monotonicity of the cost function $f()$. To overcome this problem we assign only cost to a terminal constraint if the position of the terminal is known and the terminal will not be moved anymore during the rest of the

placement. If the exact position is not known the cost for the terminal constraint are 0.

The position of the terminal is know in the following cases :

1. the terminal interval lays to the left of the cut-position and the terminal is only connected to modules in the set of not-placed modules. See figure 6.3.
2. the terminal interval lays the left of the cut-position and the terminal has is connected to at least one module in the set of of placed modules and at least one connection to set of not-placed modules. See figure 6.4.
3. the terminal has only a connection to modules in the set of placed modules. See figure 6.5.

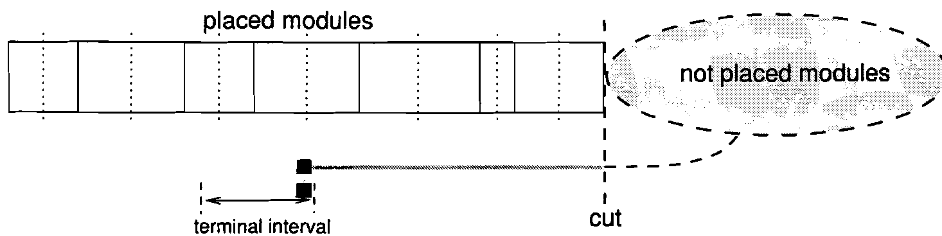

Figure 6.3: Terminal constraint cost estimation due nets connected to not-placed modules

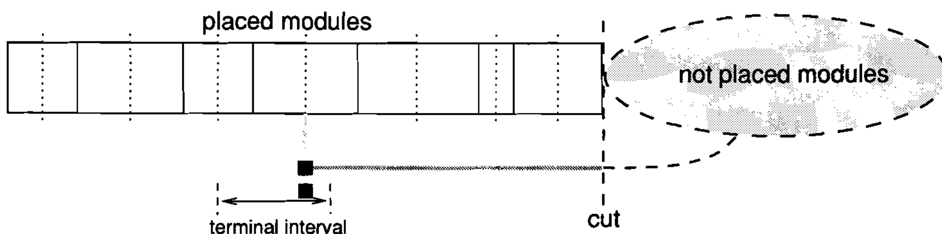

Figure 6.4: Terminal constraint cost due to nets connected to placed and not-placed modules

6.3 Modifications to the A* search

A placement program based on a plain A* search would always try to find the best placement. But the NP-completeness of the placement problem implies that in the worst case (all the nodes in the state space are visited), the solution will not be found within exponential time. Of course it would be more practicable to search for a solution path of *reasonable cost*, provided the computation required for the search is reduced. The decision of what a reasonable cost is left to the user of the program. Therefore some modifications to the A* has been made to search for a *near optimum*, decrease the computation time, or limit the amount

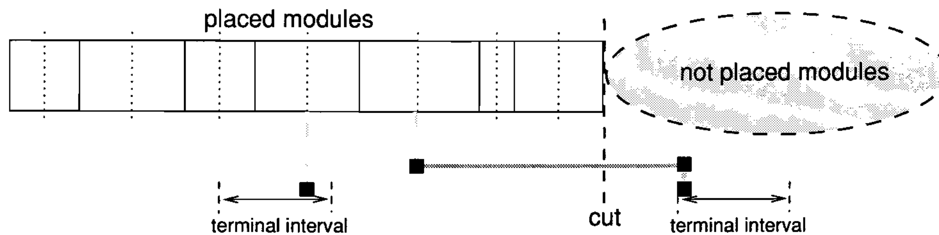

Figure 6.5: Terminal constraint cost due to nets connected to placed modules

of used memory.
Below some of the most important modification are explained.

6.4 The reduced successor set

Each time a node is expanded, the successors nodes are calculated. To speed up the A* search the successor set should be as small as possible, without leaving out important successors that could lead to the optimal placement. Under each successor of level n there are about $(n - level)! \cdot e$ unexpanded states. Thus we want the successors set as small as possible, to cut-off search paths. We could take all the not placed modules as the successor set of a node. But as we have seen in the FDP clustering, for the density minimization the successor set should only contain the the modules adjacent to the already placed modules.

To be sure that no allowed placements are left out, it has to be possible that a module with a net-delay constraint or a terminal constraint can be placed on every position in the placement. Thus the successor set of a node s, containing the set of placed modules, consists of :

1. all the modules with connections to the set of placed modules;

2. all the modules connected to a net with a net-delay constraint;

3. all the modules connected to a terminals with a terminal constraint.

If this successor set should be empty then all the not-placed modules are selected as successors.
If the number of constraints is not too large, and the modules have not many connections than the limitation of the successor set can be very effective. Especially at low levels of the search tree, when not many modules have have been placed, the successor set is very small. Here number of paths and the number of states that is cutt-off is huge. For example, in a linear placement with 20 modules there are under each successor node of level 2 about $(20-2)! \cdot e = 1.74 \cdot 10^{16}$ nodes. This shows that a lot of speedup can be achieved by keeping the successor set as small as possible.

6.5 C-algorithm

The C-algorithm (see [Goto77]) is a constructive straight-forward algorithm proposed to find a an upper bound for the cost of the linear placement. It will be used to calculate an upper-bound on the net-density, and give useful information for the ϵ-algorithm (described below). If a constraints are involved it is possible that the C-algorithm doesn't find an allowed placement, if this is the case then no solution will be given. It is merely for fast net-density minimization.

The C-algorithm calculates an upper bound for ϵ which can be used in the ϵ-algorithm. This upper-bound for ϵ gives the user of the tool a good upper-bound on the cost of the placement.

Calculation of ϵ:

$$\begin{aligned} \epsilon &= \frac{f(\text{solution}) - f(i)}{f(i)}, \\ &\text{with } f(i) \leq f(j), \forall j \in V, i \in V \\ &V = \text{set of all visited nodes} \end{aligned} \tag{6.8}$$

The C-algorithm uses a modified DFS strategy. It expands all the successors of a node. Then the cheapest of these successors is selected to be the next node to expand. The successors than haven't been selected are removed. The search is repeated with the selected node. No backtracking is used.
The time complexity of the C-algorithm is $\mathcal{O}(n^2)$.
We can modify the A*-search a bit to let it behave like the C-algorithm. The only thing to do is to clear the entire OPEN list just after a node is selected for expanding. The C-algorithm is in fact a special case of the A*-algorithm were the OPENlist can only contain one node.

The solutions found with this algorithm are mostly far from the optimum, but for circuits with only two or three wires per module and without constraints, the found net-density is almost as good as the one found with an uniform-cost search. The C-algorithm though needs, in most of the placements, far less run-time than the uniform-cost search. But the main disadvantage of the C-algorithm is that as soon as constraints are involved it is possible that this algorithm won't find an allowed placement at all.

properties of the algorithm :

- not complete
- not optimal
- fast
- useful to calculate an upper bound for the net-density

6.6 Epsilon algorithm

The ϵ-algorithm (see [Goto77]) is an approximation algorithm which produces an allowed solution whose cost is less than $(1+\epsilon)$ times the cost of the optimum solution. The algorithm

is in fact a derivation of the A* algorithm, but with a slightly different heuristic to select the next node to expand. But the price paid for the not finding the optimum solution is earned back in the form of a much faster and less memory consuming algorithm than the pure A* search. The amount of visited nodes reduced and only a portion of the search tree is expanded.

properties of the algorithm :

- complete
- not optimal

The only difference from the A* algorithm in the previous chapter is the selection of the next node to expand. Here the line :

```
n = OPEN.front();
```

is replaced by a mechanism that selects the next node from a subset of OPEN, such that the next node to expand is the closest to a solution. The subset of OPEN-list from which the node is selected contains all the nodes of which the cost are less than (1+ϵ) times the cost of the cheapest node on the OPEN-list.

$$f(u) \leq f(p) \leq (1+\epsilon) \cdot f(u), \text{ with } u \text{ the cheapest node on OPEN-list and } 0 \leq \epsilon \tag{6.9}$$

The mechanism to select the next node to expand:

```
State p = OPEN.front();
double maxcost = p.cost * (1.0 + epsilon);

do {
   if ( p.level > p2.level)
      p2 = p;
   OPEN.next(p);
} while ((p != NULL) && (p.cost <= maxcost));
// p2 is now the State to expand
```

Although in most of the circuits this algorithm will only search small parts of the search tree. In the worst case, the whole tree may have to be constructed. Which gives the algorithm the same worst case time complexity as the plain A*-algorithm. The time complexity $\mathcal{O}(n! \cdot \log n)$ For placement of most of circuits this time-complexity is far too pessimistic. The value of ϵ is very different for every circuit. There is a correlation between the computation time, the used memory and the value of ϵ. The smaller ϵ, the more time and memory is needed to find a solution. A smaller ϵ mostly result in a cheaper solution. But:

$$\begin{array}{rcl} \text{Although } f(goal_1) \leq (1+\epsilon_1) \cdot f(optimum) & \text{and} & f(goal_2) \leq (1+\epsilon_2) \cdot f(optimum), \\ 0 \leq \epsilon_1 < \epsilon_2 & \not\Rightarrow & f(goal_1) < f(goal_2), \text{ with} \\ \epsilon_1 & \rightarrow & f(goal_1); \\ \epsilon_2 & \rightarrow & f(goal_2); \end{array} \tag{6.10}$$

The ϵ determines the behaviour of the ϵ-algorithm drastically. There are two special case for the value of ϵ:

- $\epsilon = 0 \rightarrow$ the ϵ-algorithm is equivalent to the A*-algorithm
- $\epsilon = \infty \rightarrow$ the ϵ-algorithm is equivalent to the C-algorithm

If HOPPER is used as an C-algorithm it will produce an upper-bound for ϵ, lets call it ϵ'. The ϵ-algorithm will not find a cheaper solution than the one found by the C-algorithm by specifying any $\epsilon \geq \epsilon'$. (see [Goto77])

6.7 The complex OPEN list

The A* search requires a sorted priority list, to keep a list of unexpanded nodes. The priority list used in HOPPER is not a simple priority list as used in the original algorithm. Since it is not correct to compare cost of nodes of a different depth in the search tree, a number of sorted lists is used. Each list contains only nodes of the the same depth in the search tree.
In a placement with n modules there are $n + 1$ sorted lists. Another sorted list contains the $n + 1$ front nodes from the sorted lists. In figure 6.6 the structure of the OPEN list is shown. This more advanced list structure has some advantages compared to a simple sorted list :

- The lists are smaller $\rightarrow$ faster searching, sorting, adding and deleting of nodes in the lists;
- Nodes of the same level can be compared. This makes it possible to reduce the number of elements in the lists, according to the variance of cost of the nodes in the list. This will be explained in the next section;
- Faster selection of the next node to expand, with the ϵ-algorithm.

6.8 Staged search

To limit the amount of memory used by the A* algorithm, the length of the OPEN list can be limited. This is also called *staged search*. Limiting the priority list implies that some unexpanded nodes are removed from memory and that the A* algorithm is no longer optimal nor complete. Thus reducing the OPEN list can be very dangerous in combination with constraint placement. Some of the unexpanded and deleted nodes could have led to the only allowed placement, which will now never be found.
During the expansion of a node the cost of each its successor is evaluated. If a successor violates a constraint it will not be placed on the OPEN-list. This method prevents unnecessary growth of the OPEN-list. This method of immediate termination of useless successors is optional in HOPPER.
Staged search however can be very useful for net-density minimization only. It will speed up the algorithm since the number of search paths is reduced.

We use two kinds of staged search :

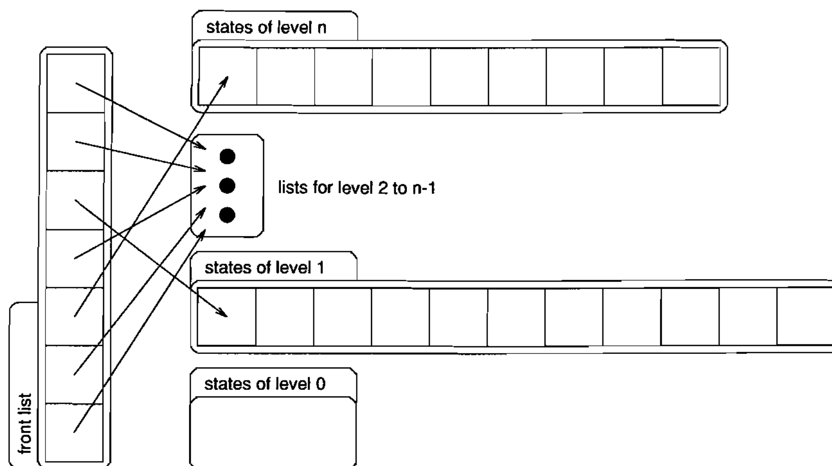

Figure 6.6: OPENlist structure

1. Allow only a maximum number of nodes on the OPEN list.
2. Let the number of nodes in the OPEN list depend on the *variance* of the nodes on the list.

OPEN list reduction using the list variance

The use of the *variance* of cost of all the states in the OPEN list can be very useful to reduce the length of the OPEN list. If the variance of the list is too high, states at the end of the the list are removed from the list.
The variance gives a measure for the spread of the cost of the states n the OPEN-list. States that have a very high cost, compared to the cheapest state, will probably be left on the list unexpanded. These states only use up memory and slow down the sorting of the list. If we chose the maximum variance wright then these states will be removed from the OPEN list without to much effect on the quality of the solution. It is however very difficult to determine what value should be used for the variance to get an acceptable solution.

Here the complex OPEN list shows its use. If the list reduction was used on a single sorted list where all the states of different level are mingled, then the list reduction would throw away many good solutions. Because states of higher levels (more modules placed) mostly have higher cost functions and thus are placed at the end of the list. In the complex OPEN list though there is a list for each level in the search tree. Thus now the variance reduction can be applied to each of these lists. Now we don't throw away to much good solutions since we only calculate the spread of states of the same level.

6.9 Iterative density reduction

One of the main disadvantages of the algorithms described above is that if a calculation has started it often takes very long before a solution is found, or even no solution at all is found.

But if we terminate the placement process and restart it with some other options, we lost valuable time, and there is still no placement.

Each time HOPPER is started with other parameter in order to decrease the net-density, the whole search tree has to be exploded from scratch, wasting time expanding the same nodes again and again.

The iterative density reduction is a method to automatically search for the minimum net-density. The constraint cost are here of second priority. The algorithm starts with a C-algorithm to determine the upper bound on the net-density. Then the net-density to search for is decreased after each found solution. An advantage is that all the found solutions are saved. Thus if the placement takes to long the search can be stopped and we always have the previous found solutions.
If a placement of a certain density has been found all the nodes with a higher or equal *net-density* are removed from the OPEN list. Disadvantage thought is that the found solutions are not optimal.
The flow-chart of the iterative density decrement control if showed in figure 6.7.
This algorithm is of high practical use. Since it is fast and very easy to use.

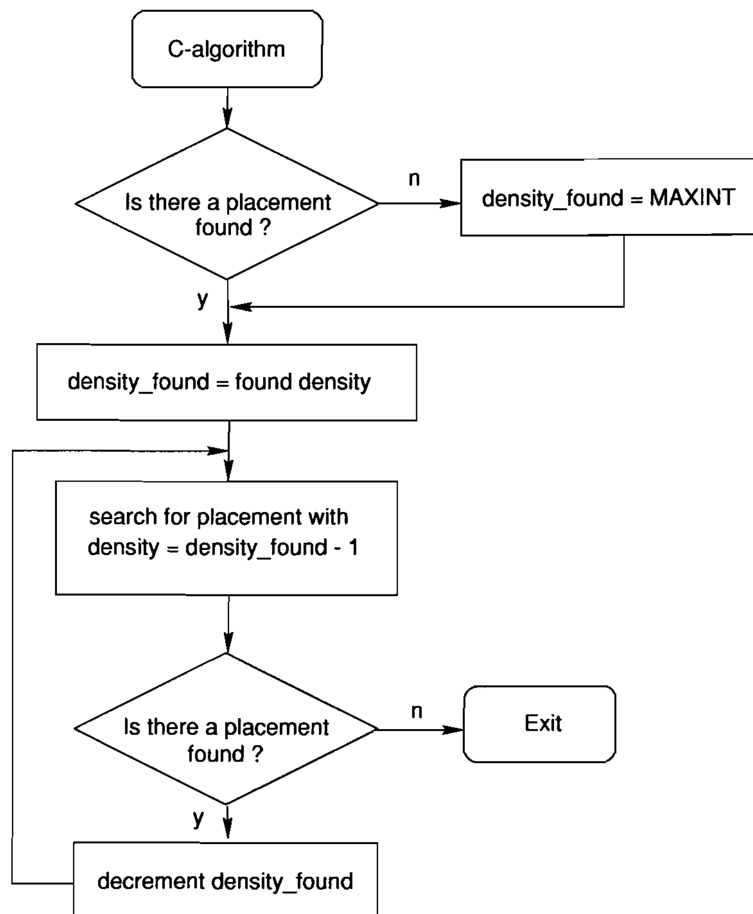

Figure 6.7: Iterative density decrement control

Chapter 7

Related Topics

The topics mentioned in this chapter are open for further research. They haven't been used in the development of HOPPER. Although they are only based on simple ideas, they could be of some use for future work.

7.1 Circuit structure

Most VLSI circuits have a similar structure. These circuits are made up of blocks of combinatorial logic and registers. The combinatorial logic blocks are separated by registers such that intermediate result can be stored in these registers. With such a *pipeline* structure a new calculation can be started even before the previous calculation is finished. Figure 7.1 shows this structure. In this picture the control logic is left out.

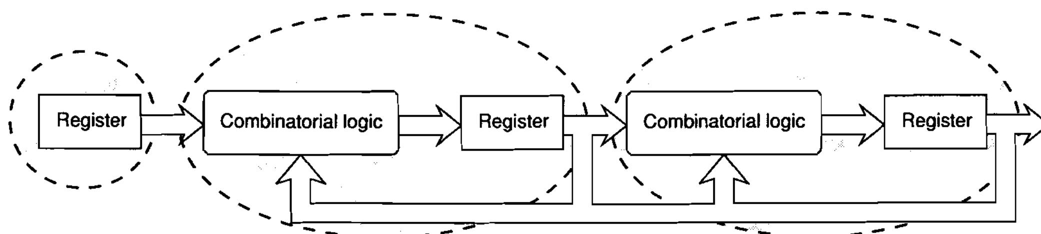

Figure 7.1: Schematic of a circuit with a pipeline structure

In most of these circuits the data-bus feedback always starts at a register and ends at a combinatorial logic block. *There is never a feedback within a combinatorial logic block.* This property can be used to place the circuit. Since the cut, through the data-bus behind a register is minimal, we can make clusters of combinatorial logic and a register (this is the gray ellipse in figure 7.1). These clusters can be placed as separate circuits. The total placement will then be a concatenation of the placements of these clusters. This *divide and conquer* approach can drastically speed up the placement generation of large circuits. But it is only valid as long as only a density minimization is concerned. As soon as terminal constraints and net constraints are involved, it is possible that this clustering makes it impossible to find an allowed placement, event if it exist.

7.2 Articulation points

As stated before it would be very useful to split the data-path into smaller parts which can be placed independently. This would speed up the placement process, but can this be done knowing that the concatenation of the partial placements is optimal or near-optimal.

Representing the network as an undirected graph, such a clustering can be made by searching for adjacent *articulation points*. See figure 7.2.

▷ **Definition 7.20.** *[articulation point]*
An articulation point in an undirected graph $G(V, E)$ is a vertex v whose removal disconnects G.

□

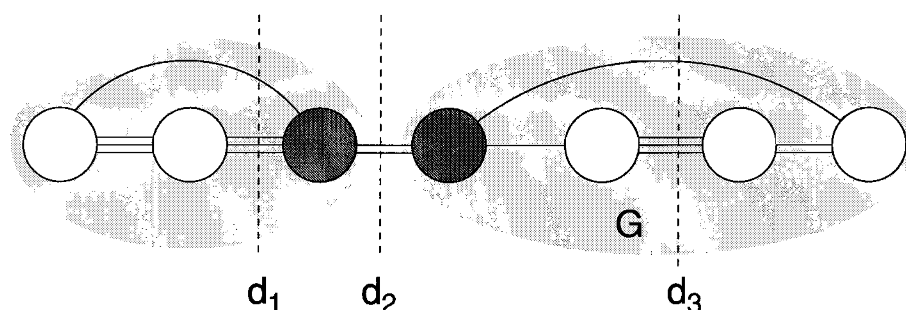

Figure 7.2: Articulation points in an undirected graph

In circuits with a pipeline structure, as in figure 7.1, the *register modules* are such articulation points. Without further proof it can be seen that the net-density d_2 is minimal. Splitting up the graph by removing the edges between two adjacent articulation points will form a clustering, while keeping the net-density minimized.

7.3 Loops

The existence of data-bus feedback implies the existence of loops in the circuit. An important property of a simple *loop* is that the linear placement can be done optimal. The minimal net-density in a *simple loop* can be easily calculated as in equation 7.1. In figure 7.3 an example of this property is displayed. See also d_1 and d_3 in figure 7.2.

▷ **Definition 7.21.** *[Loop]*
In an undirected graph $G(V, E)$, a loop is a path $p =< v_0, v_1, ..., v_k >$ of which $v_0 = v_k$ and there are at least 2 edges on the path.

□

▷ **Definition 7.22.** *[Simple loop]*
In an undirected graph $G(V, E)$, a simple loop is loop with path $p =< v_0, v_1, ..., v_k >$, and there is no sub-path of p that is a loop.

□

Each simple loop can be placed optimal such that the *minimum net-density* is :

$$d_{min} = c_{max} + c_{min} \quad \text{, with} \tag{7.1}$$
$$c_{min} = \min(i, j \in B' : |N(i) \cap N(j)|) \quad \text{,minimal connectivity in the loop} \tag{7.2}$$
$$c_{max} = \max(i, j \in B' : |N(i) \cap N(j)|) \quad \text{,maximum connectivity in the loop} \tag{7.3}$$

with B' the set of modules in the loop.
The basic principle of this property is that highly connected components should be placed adjacent to one another, thus if the components that are the least connected are placed at the most left and the most right place, the net-density will always be minimal.

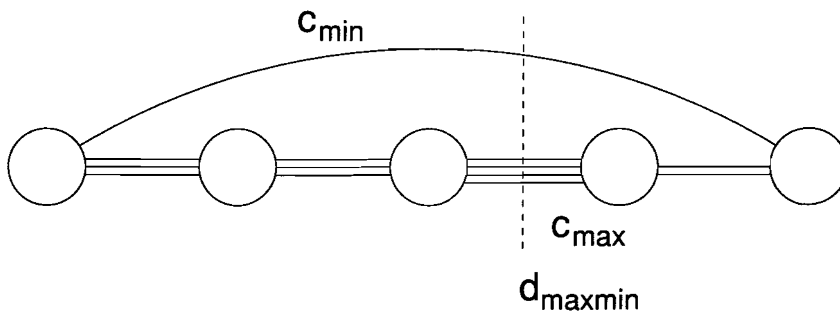

Figure 7.3: A simple loop in a network

Although most of the loops in real data-paths are simple loops, it is possible that there are intersecting loops. Intersecting loops, are loops that contain sub-loops, or are contained within another loop. Here the minimum net-density isn't as easy to calculate as for a simple loop.
If we use constraints in our placement, it is often not likely to place the loops minimizing the net-density. Therefore equation 7.1 has few practical use. Moreover it is quite expensive to search for the correct loop structures in a graph, since the number of sub-loops increases enormously as soon as there are intersecting loops. The loop properties showed in this chapter are only useful for placements with few or no constraints, that is where the net-density minimization is the major goal.
The loop properties as stated above can be useful in the estimation of a lower bound on the net-density. This can be used by placement algorithms as an initial bound to search for. This can speed up the search process significantly since no unnecessary time is wasted, searching for net-densities values that do not exist.

Chapter 8

Results

In this chapter some results are presented obtained by the program HOPPER. There have been several runs of example circuits on a HP 9000/735 workstation. The example circuits are all bit-slices of benchmark circuits. The bit-slices have been generated by a regularity extraction program.
The circuits contain different numbers of modules, wires and terminals. An overview of the test circuits is given in table B.1. In the following tables the results of of the placements of the circuits are displayed. Each table displays the performance, that is the quality and the speed, of different algorithms and options in HOPPER.
It is possible to use a combination of options. This combination of options can be used to decrease the run-time. We can for example use the iterative density reduction algorithm in combination with the reduction of the OPENlist.

The test circuits (bit-slices) are of very different structure. The test circuits have been generated by a data path regularity extraction tool. In table B.1 on overview of the size of the used test circuits is displayed.

Note : The symbol # in the tables is used to information that was not available. The information can be calculated, but for practical reasons we limited the maximum run-time of the program to 3600 seconds per placement.

8.1 Circuits without constraints

The following results are achieved by running HOPPER with different options. For the test runs in this section *no constraints* were specified. The balancing factors in the cost function have been set to $k_1 = 1, k_2 = 1, k_3 = 1$.

8.1.1 Plain A*-algorithm versus C-algorithm

To compare the several option with each other we first calculate the best placement with the plain A*-algorithm. To start the A*-algorithm just use HOPPER without extra options :
```
hopper <netlist>
```
The C-algorithm can generate very fast an upper-bound of the cost of the placement. Since we use no constraints in these tests, the cost of the placement is here equal to the net-density

in the placement.
To start the C-algorithm use HOPPER with the option -C :

```
hopper <netlist> -C
```

The results are displayed in table B.2.

- The run-times of the A*-algorithms in column 5 show that the run-time strongly depends on the structure and connectivity of the circuit
- We can see that the C-algorithm uses far less run-time to generate a placement, which is mostly of far less quality compared to the one produced by the A*-algorithm.

8.1.2 ϵ-algorithm

The ϵ-algorithm has been used here to test if the upper-bound on the net-density (in fact the total cost) calculated by the C-algorithm is correct. The C-algorithm produced a value for ϵ that is used in the ϵ-algorithm. The ϵ-algorithm will not find a placement with lower cost than the those found by the C-algorithm if an ϵ is chosen which is bigger than the ϵ value produced by the C-algorithm.
This can be very useful to get very fast some indication of the range for ϵ, which can be very different for different circuits.
If we don't know the range for ϵ and we choose a random number, that accidently is far to small, then it is possible that the run-time of the ϵ-algorithm is very long (maybe several days). The C-algorithm on the contrary will give the user very fast an upper-bound for ϵ. The user can then start several runs, while slowly decreasing the ϵ. To start the ϵ-algorithm use HOPPER with the option -e<epsilon> :

```
hopper <netlist> -e<epsilon>
```

The results are displayed in table B.3.

- We see that the net-density found by the ϵ-algorithm is not worse than the one found by the C-algorithm, while using the ϵ provided by the C-algorithm. This is conform the theory.

8.1.3 OPENlist length reduction

The OPENlist length reduction, also called staged search, is selected with the option -o<length> :

```
hopper <netlist> -o<length>
```

The results are displayed in table B.4.

- We can see that if we choose a OPENlist length of 1, then the algorithm has exactly the same behaviour as the C-algorithm (see table B.2).
- In the rows for slice Slice_AW[0]_12 and Slice_AW[0]_30 we can see that an increase of the OPENlist length can improve the quality of the placement. We see here an improvement from a net-density 9 to a net-density 6 if we OPENlist length is increased from 2 to 5.

- Comparing the run-times and the number of visited nodes we see that in most of the cases the run-time increases if the length of the OPENlist is increased.

- If the net-density found with an OPENlist length of 5 is compared to the net-density found by the plain A*-algorithm (OPENlist length ∞) (table B.2) then we see that the net-densities are almost equal, while the run time needed by plain A*-algorithm if much higher. For example for Slice_AW[1]_13 the A*-algorithm needed 2905.88 seconds, with an OPENlist length of 2 the algorithm needed only 1.53 !!! seconds to find a placement with the same net-density.

8.1.4 Iterative density reduction

To start the iterative density reduction (IDR) use HOPPER with the option -b[<minimum_density>] :

```
hopper <netlist> -b[<minimum_density>]
```

The program will stop as soon as a placement with a net-density of <minimum_density> is found.

The results are displayed in table B.5.

- The algorithm with IDR has no restriction on the length of the OPENlist. Therefore the algorithm will find an allowed placement if it exists.

- The IDR can be very successful in the search for the minimum net-density. If we compare the run-time needed to find the minimum net-density for circuit Slice_AW[0]_30, with the time needed by the plain A*-algorithm (table B.2) we see a big improvement. The A*-algorithm needs 1044.12 seconds while the IDR method find the optimum in 54.20 !! seconds.

- A great advantage of the IDR method is that the all the intermediate found placements are saved. If the search for the optimum would take to long, the program can be stopped and the best found placement can then be used.

8.2 Circuits with constraints

The following results are achieved by running HOPPER with different options. For the test runs in this section *constraints* were specified. Each circuit has 3 net-delay constraints and 3 terminal constraints. The balancing factors in the cost function have been set to $k_1 = 1, k_2 = 1, k_3 = 1$.
In the tables displayed below, the three parts of the cost are separated displayed. The total cost are the summation of these three partial costs. The unity of the displayed cost is arbitrary. The cost value can only be used to compare placements of the same circuits.

To get interesting results we allowed the program to violate the net and terminal constraints. This is done with the options: `-n -t`

The circuits used are a selection of those in table B.1. They have been selected for their not to small net-density.

An exaple of the constraint definition in a netlist:
The constraints definition in the netlist of circuit Slice_AW[6]_2 looks like :

```
TERMINAL p1
PORT po1 R0.1-0.2
TERMINAL p2
PORT po2 R0.45-0.55
TERMINAL p3
PORT po3 R0.8-0.9

NETS1435
PARAM MAXLEN 20
PIN S1435 C969 I2
PIN S1435 C982 O1

NETS1006
PARAM MAXLEN 20
PIN S1006 C378 I2
PIN S1006 C388 I2
PIN S1006 - p2

NETS343
PARAM MAXLEN 30
PIN S343 C378 I3
PIN S343 C388 I3
PIN S343 C395 I1
PIN S343 C901 O1
PIN S343 - p8
```

8.2.1 Plain A*-algorithm

To compare the several option with each other we first calculate the best placement with the plain A*-algorithm. To start the A*-algorithm just use HOPPER without extra options :
`hopper <netlist>`

The results are displayed in table B.6.

- None of the circuits could be placed without violation of the net-constraints. Since we defined $k_2 = 1$ and $k_3 = 1$, a terminal cost of less than 1 means that no terminal constraint was violated and the placement is allowed. If the terminal (constraint) cost are larger than 1 than it is not necessarily a constraint violation since it is a summation of the cost of the three terminals.

- The placement of circuit Slice_AW[6]_2 has now a net-density 4. the palcement of the circuit without constraints had a net-density 3 (table B.2.
- Slice Slice_AW[0]_12 took now 3614.72 seconds to place with net-density 4. Without constraints it took only 3.41 seconds to find a placement with net-desity 4. The introduction of constraints can increase the run-time needed to find a placement.

8.2.2 C-algorithm

To start the C-algorithm just use HOPPER with the -C option :

```
hopper <netlist> -C
```

The results are displayed in table B.7.

- Slice Slice_AW[6]_2 and Slice_AW[6]_5 are placed with a net(-delay) cost of respectively 50 and 41. This are definite not allowed placements. This shows that the C-algorithm is in most of the cases not usable for the placement of circuits with constraints.

8.2.3 ϵ-algorithm

To start the ϵ-algorithm use HOPPER with the option -e<epsilon> :

```
hopper <netlist> -e<epsilon>
```

The results are displayed in table B.8.

- From the placement of slice Slice_AW[1]_18 we see that it the decrease of ϵ not necessarily leads to an increase in run-time. If place with $\epsilon = 3$ the placement event took more than one hour, after that time the placement was stopped. With an $\epsilon = 2$ though a placement was found in 1405.49 seconds. From this we may conclude that the choice of the value for ϵ is very difficult and can only be found by *trial and error.*
- The use of the ϵ provided by the C-algorithm, these are the largest epsilons in the table, produce placements with a cost that is less or equal to the ones found by the C-algorithm.

8.2.4 Iterative density reduction

To start the iterative density reduction (IDR) use HOPPER with the option -b[<minimum_density>] :

```
hopper <netlist> -b[<minimum_density>]
```

The program will stop as soon as a placement with a net-density of <minimum_density> is found.

The results are displayed in table B.9.

- The placement of slice Slice_AW[6]_2 could be done with a net-density 3. The net cost are then very high. This shows that the IDR method does not find the optimum solution. The algorithm searches only for a placement with a net-density that is as low

as possible.
This method could be used without the option -n. Then the program will only search for placements without net-delay constraint violations.

8.2.5 OPENlist length reduction

The OPENlist length reduction, also called staged search, is selected with the option -o<length> :

```
hopper <netlist> -o<length>
```

The results are displayed in table B.11. The same results are displayed in figures.

- The placement of slice Slice_AW[0]_31 with a OPENlist-length of 10 took 1628.98 seconds. If we compare this with the optimum placement from table B.6, we see see that this took only 65.35 seconds. Thus the reduction of the OPENlist length does not automatically decrease the time necessary to find a placement. This behaviour can be explained by the fact that in the OPENlist reduction algorithm some "good" nodes are deleted. These "good" nodes would have led much faster to a better placement.

- In figure B.1 we see that for slice Slice_AW[6]_2 the increase of the OPENlist length lowered the cost of the found placement. But in figure B.2 we see that the it also took more time to find these placements.

- In figure B.1 we see that for slice Slice_AW[6]_5 the cost dramatically decrease as soon as a OPENlist larger than 1 is selected. For this slice the C-algorithm (A*-algorithm with OPENlist length 1) would be very ineffective.

- Figure B.3 shows that for slice Slice_AW[0]_31 the run-time drastically increases if the OPENlist has a length larger than 9.
For the main part of the circuit we see that the longer the OPEN-list, the more run-time is needed. The effect of the increase in run-time decreases as the OPENlist becomes longer.

- Figure B.4 shows that in some cases the increase of the OPENlist length can lead to a decrease in run-time.

- IF we compare the figures B.2, B.3 and B.4 with the figures B.5, B.6 and B.7 then we see that the run-time is almost proportional to the number of visited nodes. This means that there is a very strong correlation between the number of visited nodes and the run-time.

8.3 Summary

From the results above we can summarize the following behaviour of the different algorithms.

- The **A*-algorithm** is successful in the search for the placement with the cheapest cost. The run-time strongly depends on the connectivity of the circuits and the specified number of constraints. The A*-algorithm used on circuits with a high connectivity needs often very long time to find the optimum placement.

- The **C-algorithm** is very useful to give very fast an upper bound on the net-density. The C-algorithm can also be used to give an upper bound for the ϵ used by the ϵ-algorithm.

- The **OPENlist length reduction** mechanism can be useful to reduce the amount of used memory. It can be used to search for a near optimum placement. For most of the placements we see that the time needed to find a placement increases with length of the OPENlist. The cost of the found placement decrease with the increase of OPENlist length for most of the placements. There can however be very unpredictable behaviour in the run-time and the cost in relation to the OPENlist length.

- The **iterative density reduction** mechanism is very fast and successful for net-density minimization. In most of the cases this method is much faster than the plain A*-algorithm. The IDR mechanism is very practicable because it saves intermediate found placements. In practice this means that if the search for the minimum net-density takes to long, we can use an intermediate found placement that is almost as good as the optimal placement.

Chapter 9

Conclusions

We have seen that there are many different algorithms for the linear placement problem. But only few of those algorithms are suitable for linear placement with timing and layout constraints.

- We have seen that the force directed placement as proposed in chapter 4 cannot deal with a concatenation of net-delay constraints in critical paths. The forced directed placement however can be used for net-density minimization in bit-slices without constraints.
  The best-first search algorithm described in chapter 5 proved to be more suitable for the linear placement problem with net-delay constraints and terminal constraints.
- The best-first search algorithm is successful in the search for an allowed placement. The algorithm has two major advantages over standard search algorithms as DFS and BFS. First it can easily come out of local optima in the search space and secondly, it can use some problem specific estimation of the cost of the search path to the destination node. This makes the best-first search algorithm informed about the search path to come, which can speed up the search process significantly.
  The negative sides of the best-first search are the large amount of memory needed (in most of the cases) to find the optimal solution. This optimality also implies long run times due to the NP-completeness of the linear placement problem.
- The program HOPPER, based on a best-first search algorithm, delivers qualitative good placements. It can handle the net-delay constraints and the terminal-constraints very well. If there exists an allowed placement in the search space of the problem instance then the program will find it. In fact the more constraints are used the faster the program will find a allowed placement, if one exist.
- The program HOPPER can be an effective tool for the optimization of bit-slice placements, with or without constraints. It has several parameters with a predictable behaviour to speed up the search process, improve the quality of the placement, or limit the amount of memory needed. The run time for larger circuits, more than 30 modules, can be very long, which makes HOPPER only practicable for linear placement

of circuits smaller than 30 modules. The time needed by HOPPER to find a feasible placement strongly depends on the number of modules in the circuit and connectivity between these modules. Circuits with only a few nets per modules can be placed much faster than circuits that are fully connected.

9.1 Future work

- The linear placement tools HOPPER is quite complete with respect to the number of different options to guide the linear placement process. To make it possible to use the generated linear placements in combination with other placement tools, we need several conversion programs which can convert the output file generated by hopper into a format which can be used by other placement tools.
- One of the major disadvantages of HOPPER is the long run-time, even for small problem instances. The run-time can be improved by using more information to calculate the value of the heuristic function $h()$. This will decrease the number of nodes that have to be visited to come to an allowed placement.
- The iterative density reduction works very well for net-density minimization. This method could also be used to automatically decrease the ϵ in the ϵ-algorithm.

References

[Doll94] Doll K.
Iterative placement improvement by network flow methods.
IEEE transactions on Computer Aided Design of Integrated Cicuits and Systems, vol. 14, (1994), no. 10, p. 1189-1200.

[Jeske94] Chrzanzawska-Jeske M. and Her S-K
I/O pad assignment for force-directed placement algorithms.
Int. journal Electronics, vol. 77, (1994), no. 4, p. 467-479.

[Mathur94] Mathur A. and Liu C.L.
Compress-Relaxation: A New Approach to Performance Driven Placement for Regular Architectures
ACM, 1994.

[Nakao93] H. Nakao et al.
A high density datapath layout generation method under path delay constraints.
IEEE Custem Integrated Circuits Conference, (1993), p. 9.5.1-9.5.5.

[Sutanthavibul93] Sutanthavibul S., Shragowitz E. and Lin Rung-Bin
An Adaptive Timing-Driven Placement for High Performance VLSI's IEEE transactions on Computer-Aided design of Integrated Circuits and System, vol. 12, (1993), no. 10, p. 1488-1498.

[Cormen92] Cormen T.H
Introduction to Algorithms
MIT Press, Cambridge, 1992.

[Eijndhoven92] van Eijndhoven J.T.J.
Ontwerp Technologie
Technical University Eindhoven, College notes 5573, july 1992.

[Shinghal92] Shinghal R.
Formal Concepts in Artificial Intelligence
Chapman & Hall computing, 2nd edition, 1992.

[Cai90] Cai H. and Note S., Six P., De Man H.
A datapath layout Assembler for High performance DSP Circuits.
27th ACM/IEEE Design Automation Conference, (1990), p. 306-311.

[Wong88] Wong D.F., Leong H.W. and Liu C.L.
Simulated Annealing for VLSI design.
Dordrecht : Kluwer Academic, (1988), p. 202.

[Sechen87] Sechen C.
An Improved Simulated Annealing Algorithm for Row-Based Placement.
Proc. ICCAD, (1987), p. 478-481.

[Garey79] Garey M.R. and Johnson D.
Computers and Intractability : A Guide to the Theory of NP-completeness
W.H. Freeman, 1979.

[Lauther79] Lauther U.
A Min-Cut Placement Algoritm for General Cell Assemblies Based on Graph Representation.
Proc. of the 16th Design Automation Conference, (1979), p. 1-10.

[Goto77] Goto S.,Cederbaum I. and Ting B.S.
Suboptimum Solution of the Back-Board Ordering with Channel Capacity Constraint.
IEEE transactions on Circuits and Systems, vol 24, (1977), no 11, p. 645-651.

[Schweikert76] Schweikert D.G
A two-dimensional placement algorithm for the layout of electrical circuits
Proc. design automation conference (San Francisco CA), pp 408-416, 1976.

[Cederbaum74] Cederbaum, I.
Optimal Backboard Ordering Throug the Shortest Path Algorithm.
IEEE transactions on Circuits and Systems, vol. 21, (1974), no. 5, p. 626-632.

[Kernighan70] Kernighan B.W. and Lin S.
An Efficient Heuristic Procedure for Partitioning Graphs.
Bell Sys. tech. Journal, vol. 49, (1970), no. 2, p. 291-307.

Bibliography

[Tsay95] Tsay Y.W. and Lin Y.L.
A row-based cell placement.
IEEE transactions on Computer Aided Design of Integrated Cicuits and Systems, vol. 14, (1995), no. 3,p. 393-397.

[Tsujihashi94] Y. Tsujihashi et al.
A High-Density Data-Path Generator with Stretchable Cells.
IEEE journal of solid-state circuits, vol. 29, (1994), no. 1, p. 2-7.

[Cheng93] Cheng C.E. and Ho C.
SEFOP: A novel ap.roach to data path module placement.
IEEE 1993

[Jeske92] hrzanzawska-Jeske M. and Her S-K
Improved I/O pad assignment for sea-of-gates placement algorithm.
Proc. of the 35th Midwest Symposium on Circuits and Systems, p. 1396-1400.

[Kleinhans91] Kleinhans J.M. et al.
GORDIAN: VLSI PLacement by Quadric Programming and Slicing Optimization.
IEEE transactions on Computer-Aided Design, vol. 10, (1991), no. 3, p. 356-365.

[Onodera91] Onodera H. et al.
Branch-and-Bound Placement for Building Block Layout.
Proc. of the 28th ACM/IEEE Design Automation Conference, (1991), p. 433-439.

[Sigl91] Sigl G., Doll K., Johannes M.
Analytical Placement: A linear or a Qaudratic Objective Function ? Proc. of the 28th ACM/IEEE Design Automation Conference, (1991), p. 427-431.

[Shiochi90] Shiochi M. et al.
New Design Approach for Configurable Data-Path.
IEEE Custom Integrated Circuits Conference (CICC), (1990), p. 14.5.1-14.5.4.

[Nakao89] akao H. et al.
A module generation system for array structure modules.
IEICE 1989 spring Nat. Convention Rec., vol 1, p. 421.

[Tsay88] Tsay R-S, Kuh E-S and Hsu C-P
PROUD: A Fast Sea-of-Gates Placement Algorithm.
Proc of th 25th ACM/IEEE Design Automation Conference, (1988), p. 318-323.

[Sechen86] Sechen C. and Sagivanni-Vincentelli A.
Timberwolf 3.2: A New Standard Cell Placement and Global Routing Package.
Proc. of the 23rd Design Automation Conference, (1986), p. 432-439.

[Hartoon86] Hartoog M.R.
Analysis of placement procedures for VLSI standard cell layout.
Proc. of the 23th Design Automation Conference, (1986), p. 314-319.

[Just86] Just K.M., Kleinhans J.M. and Johannes F.M.
On the Relative Placement and The Transportation Problem for standard-Cell layout.
Proc of the 23th Design Automation Conference, (1986), p. 308-313.

[Cheng85] Cheng C.K.
Decomposition algorithms for linear placement and application to VLSI design.
Proc. of ISCAS, (1985), p. 1047-1050.

[Shirai84] hirai Y. et al.
Artificial intelligence, Concepts, Techniques and Applications.
John Wiley and Sons, 1984

[Kang83] Kang S.
Linear Ordering and Application to Placement.
Proc. of the 20th Design Automation Conference, (1983), p. 457-464.

[Asano82] Asano T.
An Optimum gate Placement Algorithm For MOS One-dimensional Arrays.
Journal of Digital Systems, vol 6, (1982), no. 1, p. 1-24.

[Ohtsuki79] Ohtsuki T.et al.
One Dimensional Logic Gate Assignment an Interval Graphs.
IEEE transactions on Computer-Aided design of Integrated Circuits and System, vol. 26, (1979), no. 9, p. 675-684.

[Breuer77] reuer M.A.
A Class of Min-Cut Placement Algorithms.
14th Design Automation Conference, (1977), p. 284-290.

[Yoshizawa75] Yoshizawa H. et. al.
A Heuristic procedure for ordering MOS arrays.
Proc. of the 12th Design Automation Conference, (1975), p. 384-89.

[Adolphson73] Adolphsonan D. and Hu T.C.
Optimal Linear Ordering.
SIAM J. Appl. Math, vol. 25, (1973), no. 3, p. 403-423.

[Hall70] Hall K.M.
An r-Dimensional Quadric Placement Algoritm.
Management Science, vol. 27, (1970), no. 3, p. 219-229.

Appendix A

Specification of constraints

In order to specify the constraints in a net-list file, a few new parameters have been added to the existing ones. In the net-list the following additional commands can be used.

Net-delay constraints

Net-delay constraints can be specified by adding a PARAM MAXLEN command to a net section. The net-delay constraint definition :

```
<net-delay-constraint> ::= ''PARAM MAXLEN'' <number>
```

The number is the maximum length of the net in arbitrary units.

Terminal constraints

Terminal constraints can be specified by adding a PORT definition to a terminal section. The terminal range is given as a percentage of the total placement width. The range values will be mapped to positions at the bottom of the placement rectangle. The PORT definition :

```
<port> ::= ''PORT'' <portname> <position>
<position> ::= ''R'' <f> [ ''-'' <f> ]
<f> ::= unsigned floating point
```

The mapping of <f> :

$$mapping(f) = \begin{cases} f & \text{if } 0 \leq f < 1.0 \\ 1.0 & \text{if } 1.0 \leq f < 2.0 \\ 3.0 - f & \text{if } 2.0 \leq f < 3.0 \\ 0.0 & \text{if } 3.0 \leq f < 4.0 \end{cases} \tag{A.1}$$

Module width

The get a correct placement it is necessary to specify the width of a each module in the placement. Without the exact width the net-delay constraints and the terminal constraints

cannot be calculated correctly.
The width of a module can be specified by adding a PARAM WIDTH command to an instance section. The module width definition :

```
<module width> ::= ''PARAM WIDTH'' <number>
```

The number is the module width in arbitrary units.

If no width is specified, HOPPER will generate an arbitrary width for the module proportional to the number of connectors in the module.

Appendix B

Tables and figures

Table B.1: Used test circuits

circuit	#modules	#nets	#terminals
Slice_AW[0]_10	9	28	19
Slice_AW[6]_2	10	27	22
Slice_AW[6]_5	10	21	16
Slice_AW[0]_31	13	38	28
Slice_AW[0]_22	14	55	42
Slice_AW[0]_12	17	46	29
Slice_AW[0]_13	17	46	29
Slice_AW[0]_1	20	44	33
Slice_AW[0]_30	22	57	38
Slice_AW[1]_9	26	52	39
Slice_AW[1]_22	27	38	27
Slice_AW[1]_13	28	56	43
Slice_AW[1]_18	30	62	48

Table B.2: plain A*-search (optimum) versus C-algorithm

circuit	#modules	optimum			C-algorithm		
		net-density	#visited nodes	time [s]	net-density	#visited nodes	time [s]
Slice_AW[0]_10	9	1	46	0.04	1	46	0.04
Slice_AW[6]_2	10	3	4434	4.30	3	34	0.03
Slice_AW[6]_5	10	3	1191	0.80	3	33	0.03
Slice_AW[0]_31	13	2	259535	759.80	2	57	0.07
Slice_AW[0]_22	14	3	299871	2438.68	3	64	0.07
Slice_AW[0]_12	17	4	2965	3.41	9	118	0.15
Slice_AW[0]_13	17	4	2965	3.40	9	118	0.15
Slice_AW[0]_1	20	2	196	0.23	3	111	0.14
Slice_AW[0]_30	22	4	72790	1044.12	9	134	0.21
Slice_AW[1]_9	26	5	8699	14.08	6	126	0.21
Slice_AW[1]_22	27	6	296735	505.52	8	155	0.26
Slice_AW[1]_13	28	6	1163268	2905.88	8	157	0.26
Slice_AW[1]_18	30	5	21024	36.47	6	161	0.29

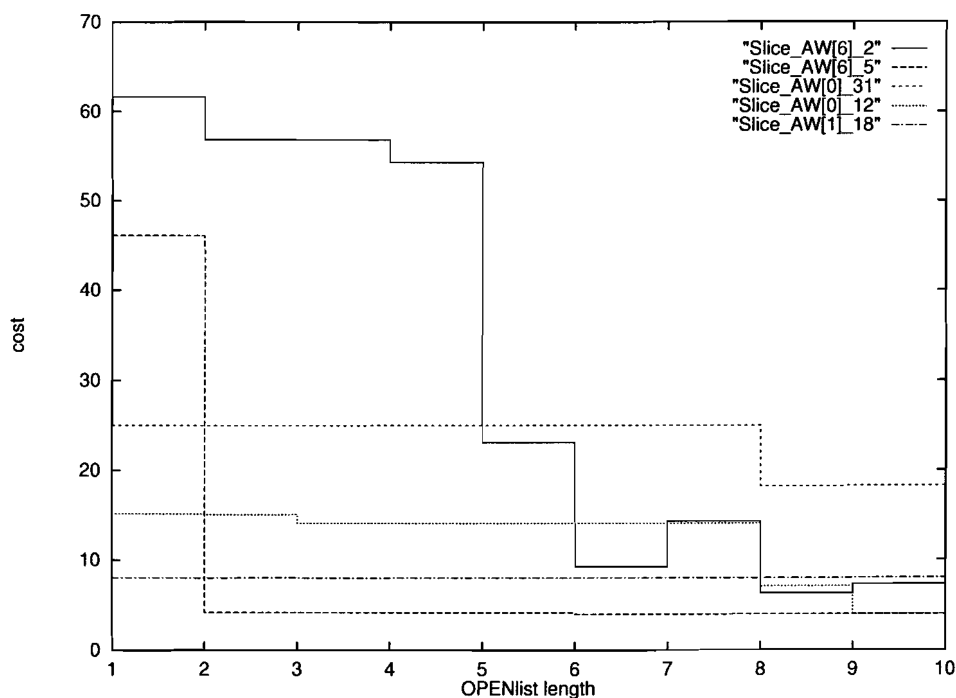

Figure B.1: OPENlist length reduction : OPENlist length versus placement cost

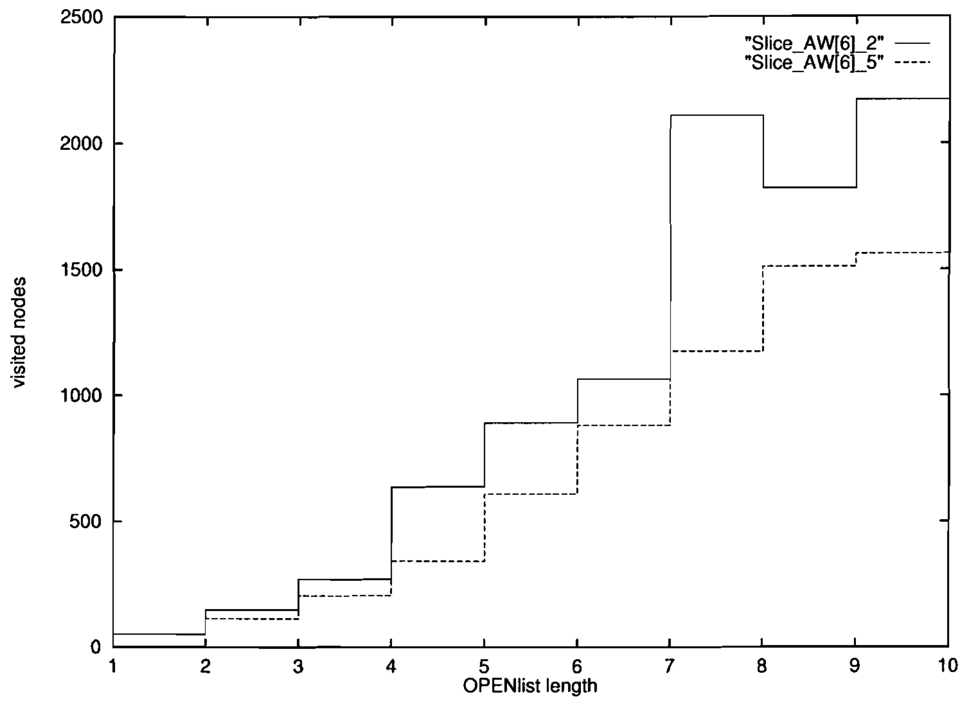

Figure B.2: OPENlist length reduction : OPENlist length versus number of visited nodes

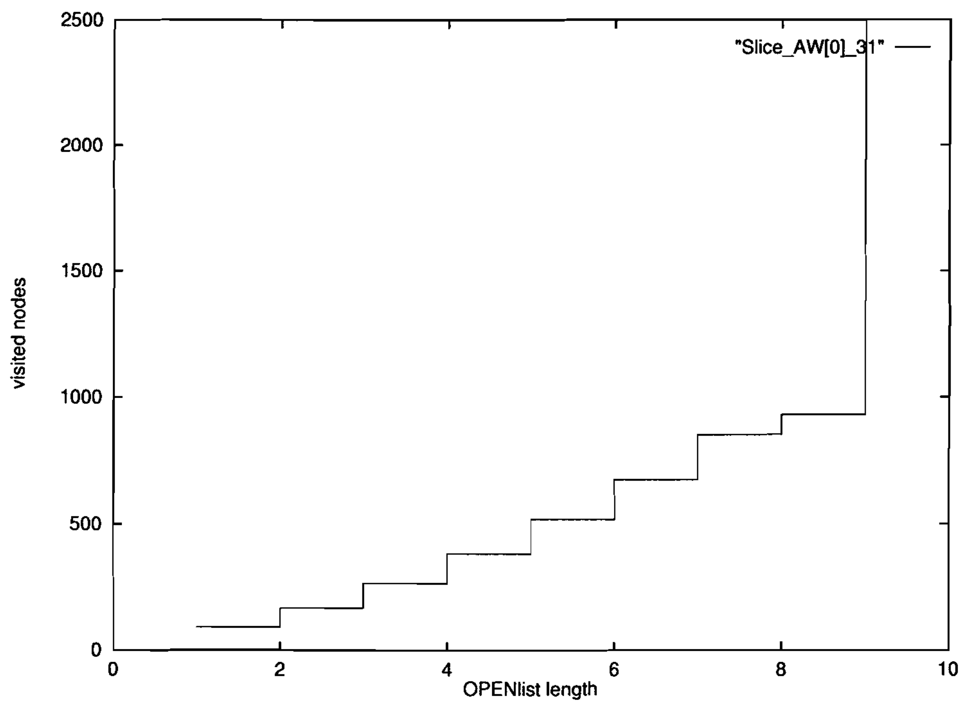

Figure B.3: OPENlist length reduction : OPENlist length versus number of visited nodes

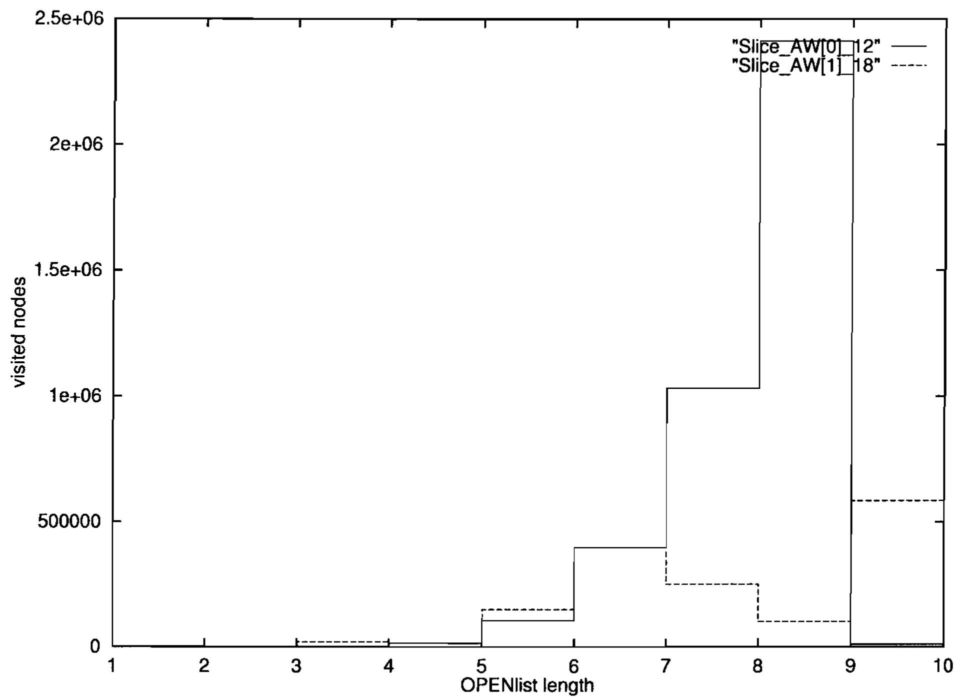

Figure B.4: OPENlist length reduction : OPENlist length versus number of visited nodestest

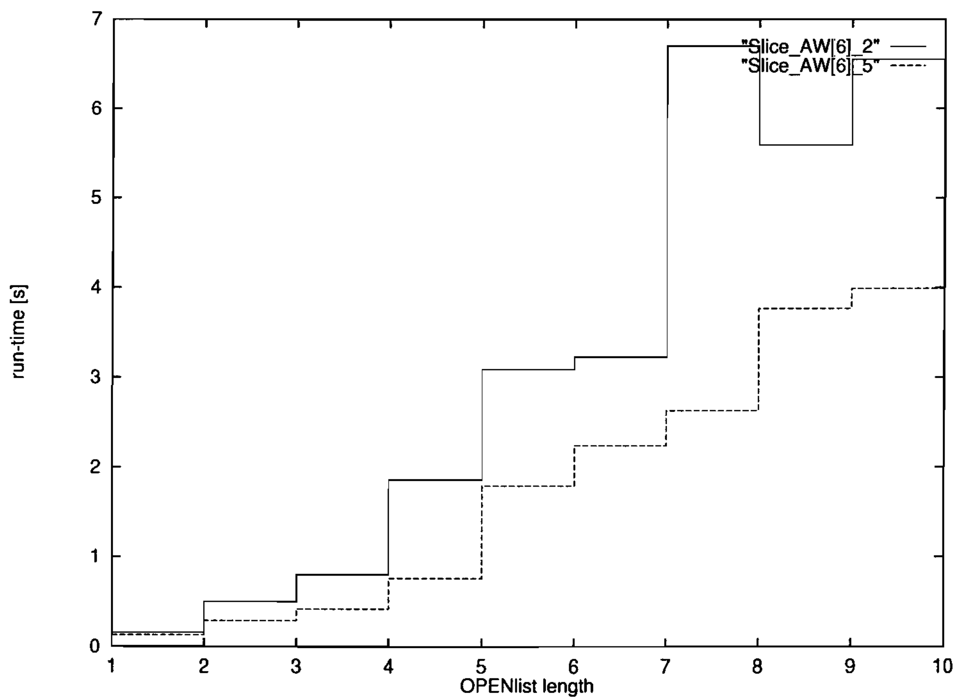

Figure B.5: OPENlist length reduction : OPENlist length versus number of visited nodes

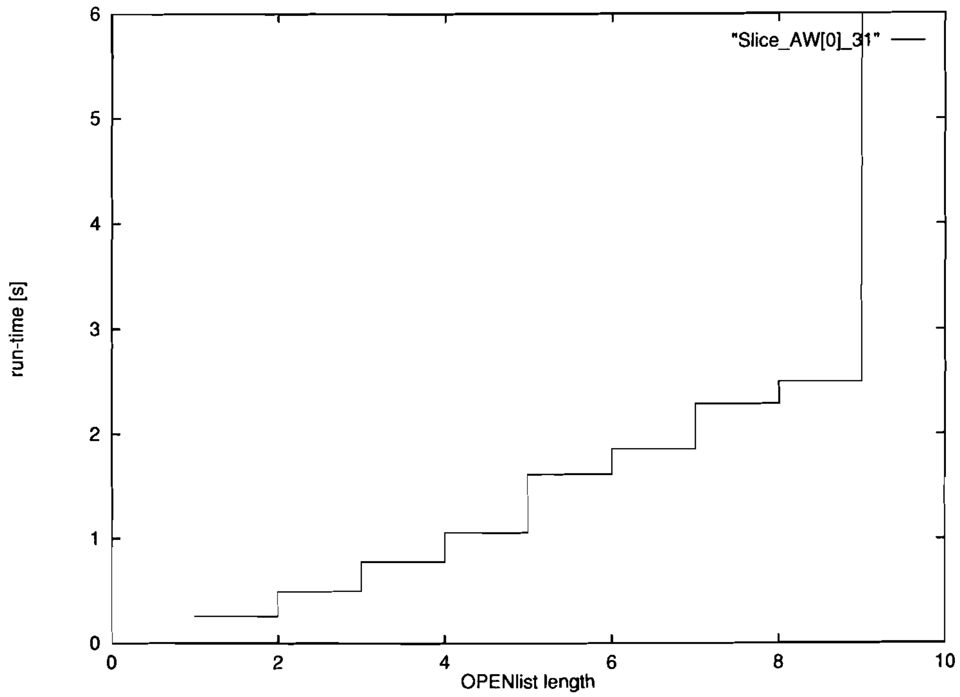

Figure B.6: OPENlist length reduction : OPENlist length versus number of visited nodes

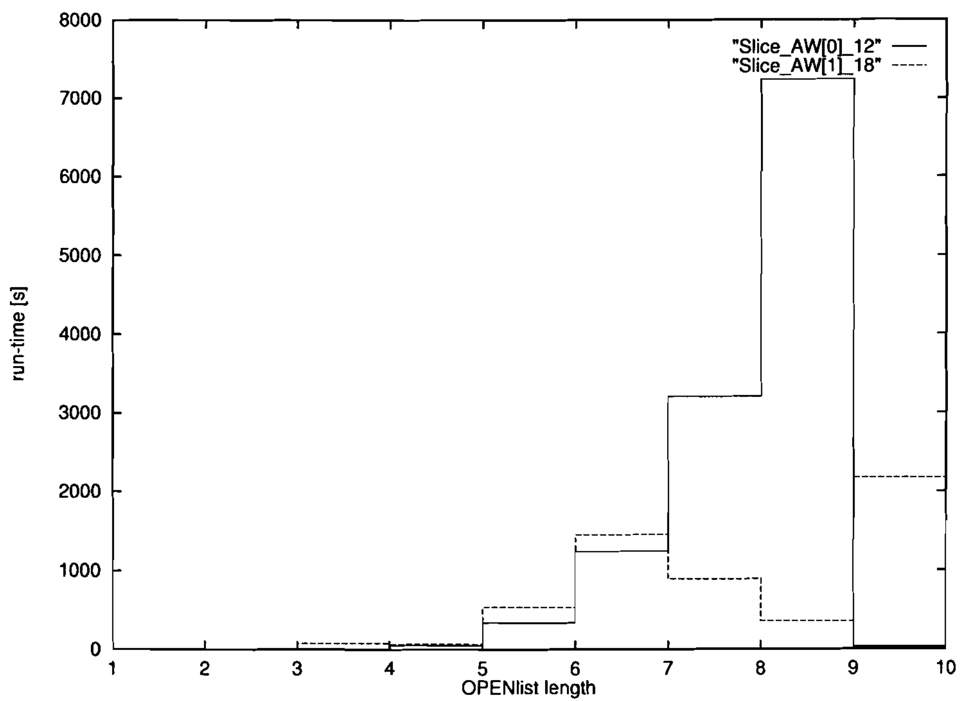

Figure B.7: OPENlist length reduction : OPENlist length versus number of visited nodes

Table B.3: Results of the ϵ-algorithm

circuit	ϵ (from C-algorithm)	net-density	time[s]	#visited nodes
Slice_AW[0]_10	0	1	0.04	46
Slice_AW[6]_2	2	3	0.03	34
Slice_AW[6]_5	2	3	0.03	33
Slice_AW[0]_31	1	2	0.06	57
Slice_AW[0]_22	2	3	0.07	64
Slice_AW[0]_12	8	9	0.15	118
Slice_AW[0]_13	8	9	0.15	118
Slice_AW[0]_1	2	3	0.14	111
Slice_AW[0]_30	8	9	0.21	134
Slice_AW[1]_9	5	6	0.22	126
Slice_AW[1]_22	7	8	0.24	155
Slice_AW[1]_13	7	8	0.26	157
Slice_AW[1]_18	5	6	0.28	161

Table B.4: Results for OPEN-list length reduction

circuit	length 1			length 2			length 5		
	net-density	time[s]	#visited nodes	net-density	time[s]	#visited nodes	net-density	time[s]	#visited nodes
S..[0]_10	1	0.04	46	1	0.04	46	1	0.04	46
S..[6]_2	3	0.04	34	3	0.32	340	3	1.30	1347
S..[6]_5	3	0.02	33	3	0.17	229	3	0.67	923
S..[0]_31	2	0.06	57	2	0.70	525	2	118.51	77295
S..[0]_22	3	0.07	64	3	0.61	471	3	91.30	65311
S..[0]_12	9	0.16	118	9	0.37	309	* 6	0.63	556
S..[0]_13	9	0.14	118	9	0.38	309	6	0.62	556
S..[0]_1	3	0.14	111	2	0.17	140	2	0.19	167
S..[0]_30	9	0.20	134	9	0.79	566	* 6	77.38	58712
S..[1]_9	6	0.22	126	5	0.49	270	5	2.52	1636
S..[1]_22	8	0.26	155	6	0.78	470	6	18.62	12696
S..[1]_13	8	0.26	157	6	* 1.53	927	6	35.41	23849
S..[1]_18	6	0.29	161	5	0.81	450	5	5.75	3390

Table B.5: Results for iterative density reduction

circuit	net-density	time[s]	#visited nodes
Slice_[0]_10	1	0.04	46
Slice_[6]_2	3	0.02	34
Slice_[6]_5	3	0.04	49
Slice_[0]_31	2	0.06	57
Slice_[0]_22	3	0.06	64
Slice_[0]_12	9	0.15	118
	8	0.25	179
	7	0.36	250
	6	0.48	340
	5	0.63	450
	4	0.80	581
Slice_[0]_13	9	0.15	118
	8	0.26	179
	7	0.36	250
	6	0.49	340
	5	0.64	450
	4	0.81	581
Slice_[0]_1	3	0.13	111
	2	0.25	204
Slice_[0]_30	9	41.07	24853
	8	41.22	24914
	7	43.62	24985
	6	50.08	25075
	5	53.81	25185
	4	* 54.20	25316
Slice_[1]_9	6	0.21	130
	5	0.37	218
Slice_[1]_22	8	0.47	300
6	7	0.66	406
5	6	0.92	578
Slice_[1]_13	8	0.62	423
	7	0.80	529
	6	1.08	701
Slice_[1]_18	6	0.62	391
	5	0.97	604

Table B.6: Results for pure A*-search (optimum) with constraints

circuit	net-density	net cost	terminal cost	time[s]	#visited nodes
Slice_[0]_10	1	0.0	0.222223	10.07	3881
Slice_[6]_2	* 4	0.0	0.330989	6.17	2304
Slice_[6]_5	3	0.0	0.689678	2.90	1186
Slice_[0]_31	3	0.0	1.003580	65.35	26503
Slice_[0]_12	4	0.0	0.024439	* 3614.72	631351
Slice_[1]_18	#	#	#	>3600	#

Table B.7: Results for C-algorithm with constraints

circuit	net-density	net cost	terminal cost	time[s]	#visited nodes
Slice_[0]_10	1	0.0	0.222223	0.14	46
Slice_[6]_2	5	* 55.00	1.625203	0.18	52
Slice_[6]_5	4	* 41.00	1.147930	0.12	53
Slice_[0]_31	4	20.0	1.012346	0.26	92
Slice_[0]_12	6	8.00	1.178496	0.52	142
Slice_[1]_18	7	0.0	1.013112	1.53	278

Table B.8: Results for ϵ-algorithm with constraints

circuit	ϵ	net-density	net cost	terminal cost	time[s]	#visited nodes
Slice_AW[0]_12	1	5	0.0	0.088863	4.34	1512
Slice_AW[0]_12	2	5	0.0	0.192501	1.34	422
Slice_AW[0]_12	3	5	0.0	2.064740	1.04	325
Slice_AW[0]_12	4	6	0.0	2.064740	0.97	301
Slice_AW[0]_12	5	6	0.0	2.064740	0.98	301
Slice_AW[0]_12	6	6	0.0	2.064740	1.07	301
Slice_AW[0]_12	7	6	8.00	1.178496	0.54	142
Slice_AW[0]_12	8	6	8.00	1.178496	0.54	142
Slice_AW[0]_31	1	#	#	#	>3600	#
Slice_AW[1]_18	1	#	#	#	>3600	#
Slice_AW[1]_18	2	7	0.0	1.013112	* 1405.49	291851
Slice_AW[1]_18	3	#	#	#	* >3600	#
Slice_AW[1]_18	4	7	0.0	1.013112	33.20	8459
Slice_AW[1]_18	5	7	0.0	1.013112	348.32	82674
Slice_AW[6]_5	1	4	0.0	0.190007	1.05	431
Slice_AW[6]_5	2	4	0.0	0.190007	0.74	308
Slice_AW[6]_2	1	4	0.0	0.330989	4.64	1600
Slice_AW[6]_2	2	4	0.0	0.330989	1.20	401

Table B.9: Results for iterative density reduction with constraints

circuit	net-density	net cost	terminal cost	time[s]	#visited nodes
Slice_AW[0]_10	1	0.0	0.222223	10.45	3881
Slice_AW[6]_2	5	44.00	0.633856	0.41	111
	4	27.00	0.936722	0.49	132
	* 3	* 50.200001	2.625202	0.55	148
Slice_AW[6]_5	4	11.00	1.805392	1.45	598
	3	0.0	1.042078	2.10	826
Slice_AW[0]_31	4	20.0	1.012346	0.56	253
	3	14.00	1.285865	0.96	424
Slice_AW[0]_12	#	#	#	>3600	#
Slice_AW[1]_18	7	0.0	1.013112	4.62	867
	6	0.00	2.000001	5.91	1112

Table B.10: Results for OPEN-list length reduction with constraints

circuit	list length	net-density	net cost	terminal cost	time[s]	#visited nodes
Slice_AW[0]_10	1	1	0.0	0.222223	0.13	46
	2	1	0.0	0.22	0.73	249
	3	1	0.0	0.22	2.43	920
	4	1	0.0	0.22	4.72	2087
	5	1	0.0	0.222	10.16	3236
	6	1	0.0	0.22	9.43	3881
	7	1	0.0	0.22	9.28	3881
	8	1	0.0	0.222	10.10	3881
	9	1	0.0	0.222	10.17	3881
	10	1	0.0	0.222	10.52	3881
Slice_AW[0]_12	1	6	8.0	1.17	0.50	142
	2	6	8.0	1.114072	5.19	1252
	3	5	8.0	1.114072	11.34	3665
	4	5	8.0	1.114072	51.23	16492
	5	5	8.0	1.114072	338.18	107972
	6	5	8.0	1.114072	1241.43	398169
	7	5	8.0	1.114072	3205.77	1033808
	8	6	0.	1.114072	7239.35	2412194
	9	4	0.0	0.024439	29.28	10245
	10	4	0.0	0.024439	34.90	12353
Slice_AW[0]_31	1	4	20.0	1.012346	0.26	92
	2	4	20.0	1.012346	0.50	169
	3	4	20.0	1.012346	0.78	266
	4	4	20.0	1.012346	1.06	383
	5	4	20.0	1.012346	1.62	520
	6	4	20.0	1.012346	1.86	677
	7	4	20.0	1.012346	2.29	854
	8	3	14.0	1.285865	2.50	932
	9	3	14.0	1.285865	13.98	4064
	10	3	8.0	1.836365	* 1628.98	437953

Table B.11: Results for OPEN-list length reduction with constraints (part-2)

circuit	list length	net-density	net cost	terminal cost	time[s]	#visited nodes
Slice_AW[1]_18	1	7	0.0	1.013112	1.44	278
	2	7	0.0	1.013112	9.72	2303
	3	8	0.0	0.001457	82.26	21222
	4	7	0.0	1.036421	67.40	18384
	5	7	0.0	1.036421	539.09	150249
	6	7	0.0	1.036421	1450.88	398478
	7	7	0.0	1.036421	893.37	251462
	8	8	0.0	0.001457	359.76	101537
	9	7	0.0	1.013112	2174.28	584603
	10	7	0.0	1.013112	2011.60	541817
Slice_AW[6]_2	1	5	55.0	1.625203	0.16	52
	2	4	50.2	2.625202	0.50	149
	3	4	50.2	2.625202	0.80	271
	4	4	48.2	2.106003	1.86	639
	5	6	16.0	1.106004	3.09	893
	6	5	3.0	1.244456	3.23	1066
	7	3	9.2	2.138453	6.70	2108
	8	5	0.0	1.311520	5.59	1824
	9	4	2.0	1.311520	6.55	2171
	10	4	2.0	1.311520	5.39	2341
Slice_AW[6]_5	1	4	41.0	1.147930	0.13	53
	2	4	0.0	0.190007	0.29	115
	3	4	0.0	0.190007	0.42	206
	4	4	0.0	0.190007	0.76	343
	5	4	0.0	0.190007	1.79	610
	6	3	0.0	1.016437	2.24	882
	7	3	0.0	1.016437	2.63	1177
	8	3	0.0	1.016437	3.77	1513
	9	3	0.0	1.016437	3.99	1564
	10	3	0.0	1.016437	4.50	1727