An Exact Algorithm for Solving Difficult Detailed Routing Problems

Kolja Sulimma
J. W. Goethe-Universität Frankfurt
kolja@sulimma.de

Wolfgang Kunz
J. W. Goethe-Universität Frankfurt
w_kunz@em.informatik.uni-frankfurt.de

ABSTRACT

Channel routing is an NP-complete problem. Therefore, it is likely that there is no efficient algorithm solving this problem exactly.

In this paper, we show that channel routing is a fixed-parameter tractable problem and that we can find a solution in linear time for a fixed channel width.

We implemented our approach for the restricted layer model. The algorithm finds an optimal route for channels with up to 13 tracks within minutes or up to 11 tracks within seconds.

Such narrow channels occur for example as a leaf problem of hierarchical routers or within standard cell generators.

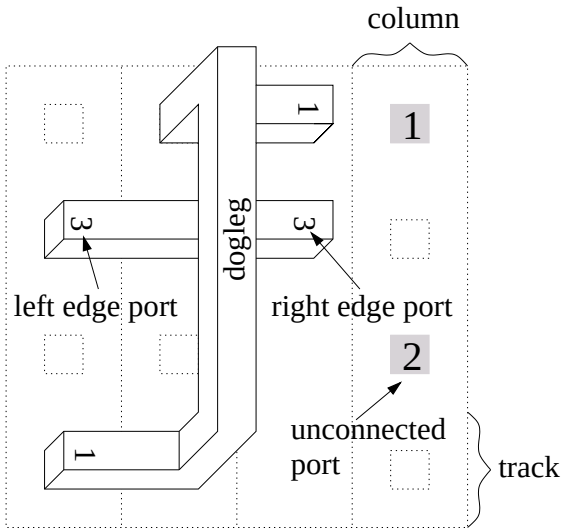

Figure 1: A partially routed four column segment of a channel with four tracks

ISPD'01, April 1-4, 2001, Sonoma, California,USA.

1. INTRODUCTION

The channel routing problem addressed in this paper is the problem of connecting ports belonging to the same net in a channel of length n.

In the following we assume grid based routing in the restricted two layer routing model. In a grid-based model, all elements are aligned with respect to a given routing grid. In the restricted layer model one layer is reserved for horizontal connections—or trunks—and the other layer is reserved for vertical connections called doglegs.

The most common formulation of the channel routing problem aligns all ports on the top and bottom boundary of the channel. We targeted the same model, but the resulting algorithm can solve more general problem instances at no additional cost in runtime or memory.

Over-the-cell-routing: the terminals of the nets can be placed at arbitrary positions within the channel. It should be noted that the placement of more than two ports on a column is likely to be unroutable without detours.

Switchbox support: the problem instances are allowed to have ports on the left and right boundary of the channel.

Obstacles: the problem instance may include obstacles of arbitrary shape in any layer. If there exists a solution to the routing problem in the given routing model, the algorithm will find it.

Single layer doglegs: although we did not test it, we believe that the routing model can be extended to allow single layer doglegs with only minor changes to the algorithm.

Our algorithm tries to route the channel with a given number t of routing tracks. This is appropriate for applications with a fixed area budget, such as standard cell generation. To find the minimum realizable channel width, the algorithm must be invoked for each width, starting with the channel density.

The main restriction of the routing model used in this paper is that each signal crosses each vertical column of the channel at most once. This means that there are no detours. A signal cannot fork and cannot be routed in an open loop.

There are straightforward extensions of our approach to multi-layer routing and for the unrestricted layer model.

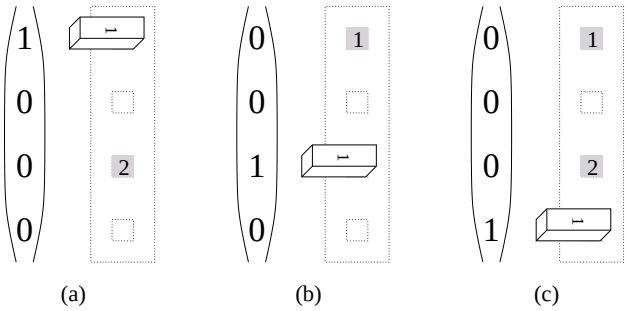

Figure 2: Signal 1 enters the last colum of the example in Figure 1 on all three possible tracks. It can not enter on the second track, because a port for signal 3 in the second column blocks that track.

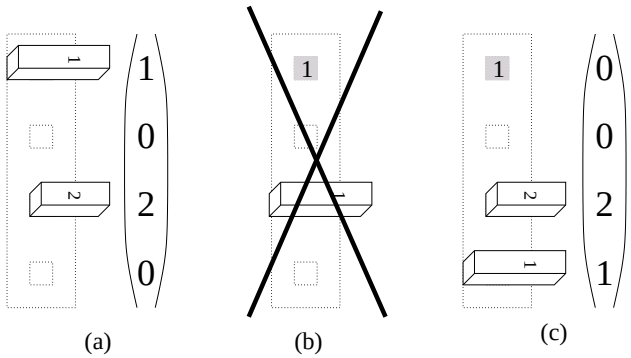

Figure 3: AddLeftEdgePorts() inserts signal 2 that starts at this column into the assignments of figure 2. Assignment (b) is rejected because port 2 would be blocked by signal 1. The assignment vectors are shown to the right because from now on we are looking at where signals are leaving the column.

For these extensions the theoretical result of linear runtime for a given channel width still holds but for our implementation the maximum tractable channel width becomes too small for practical applications.

The channel routing problem was introduced by Hashimoto and Stevens [1] in 1971. It is a well studied problem and there exist fast algorithms that provide a close to optimal solution such as the greedy router presented in [2]. However, channel routing is often used in the context of time-consuming toolflows. For example, many standard cell placers spend hours for simulated annealing and state of the art CMOS cell generators take minutes to solve integer linear programs to obtain the optimal layouts. In contexts like these it is not necessary to have very fast routing algorithms. Instead, it makes sense to spend some more computing power on routing in an attempt to further improve the quality of the resulting layout.

The main motivation for this work has been to develop a method for intra cell routing. Standard cell generators and on the fly leaf cell generation encounter small but difficult routing problems. Tools like LAS [7] from Cadence Design Systems choose channel routing for intra cell routing. Intra cell routing clearly benefits from our ability to handle ports inside the channel and obstacles such as via placement contraints from layout features of the active, poly and contact layer. Also, to find the optimal routing is very important for intra cell routing. A routing algorithm that needs an additional routing track could make it impossible to realize a leaf cell in the allocated cell height, thus deteriorating the quality of the cell library.

2. ALGORITHM

Key to the proposed algorithm is the concept of track assignments. A track assignment is an injective mapping of signals to routing tracks.

We represent a track assignment by recording for each track the signal that leaves the column on this track. Alternatively, we could record for each signal the track that it is on. There are t tracks so that a given column can only be crossed by at most t signals. It is therefore sufficient for either semantic to represent the track assignment by a t-tupel $(s_1, \ldots, s_t)$ of integers in the range $[0 \ldots t]$.

The proposed algorithm uses dynamic programming to process the channel from left to right, one column at a time. For each column the set of all track assignments for signals leaving the column is computed. This is achieved by examining each track assignment that could leave the previous column and listing all track assignments that can be reached from this assignment in the current column.

The general procedure is as follows:

```
for (each column)
   for (each possible track assignment m
        of the last column) {
      check if m is valid in this column with
         respect to obstacles and ports;
      generate new track assignments by
         adding all combinations of doglegs;
   }
```

The following code block shows the processing of one column in more detail. It takes as input a set M_{i-1} of all possible track assignments for the previous column and produces a set M_i of all possible track assignments for the current column as output. In addition, some of the subroutines need information about the port locations.

```
next_column(M_{i-1}) {
  M_i ← ∅
  for each assignment m∈ M_{i-1} {
     m ← removeRightEdgePorts(m, i-1)
     m ← addLeftEdgePorts(m, i)
     m ← connectPorts(m, i)
     m ← checkForObstacles(m, i)
     if (m ≠null) {
        M_i ← M_i∪{m}
        M_i ← M_i∪doglegs(m)
     }
  return M_i
}
```

We will use the partially routed channel of Figure 1 as an example to illustrate how the algorithm determines all the track assignments that can leave column 4 to the right. We assume that signal 1 and signal 2 have ports to the right of the shown channel segment.

Figure 2 list all the track assignments that can enter column 4 from the left. The procedure next_column() processes one incoming track assignment m at a time.

For each assignment the procedure removeRightEdgePorts() is called. It removes all signals from the assignment that do not have any ports at the current or at any of the following columns. These signals end in the previous column and do not need to be considered any longer.

Then, addLeftEdgePorts() is called. This procedure adds all signals having their first port on the current column to the assignment. This can only succeed if m has an empty track everywhere there is such a port in the column. Otherwise, *null* is returned to inform the caller that this assignment is invalid. An example is shown in Figure 3.

ConnectPorts() is called next. It checks all ports that belong to a signal that has ports to the left of the current column. If a track needed by such a port is occupied by another signal, null is returned. If the track is occupied by the signal that connects to the port, m is returned unmodified. If the track is free a dogleg is introduced to connect the correct signal to the port. Figure 4 shows how this procedure modifies the assignments of Figure 3.

Finally checkForObstacles() tests if any of the tracks used by the generated assignment are blocked by channel obstacles.

If m is a valid assignment, i.e. none of the previous procedures returned *null*, it can be added to the set M_i of valid mappings for this column. Additional valid track assignments are generated by adding all possible combinations of doglegs in procedure doglegs() as shown in Figure 5. The algorithm could easily be extended to support single layer doglegs. In the shown example an additional assignment (0, 2, 0, 1) can be generated from (a.1), (a.5) or (c.2) if single layer doglegs are supported.

2.1 Layout generation

For each generated track assignment we store a pointer to the track assignment in the previous column from which it was generated.

When the last column is reached with a nonempty set of assignments, we know that the channel is routable. Then we can select a track assignment from the last column and follow the pointers back to the first column. The required dogleg layout can easily be determined for each column from the track assignment and its predecessor.

If a track assignment is generated multiple times for a column it is stored only once in the set of value assignments although there are multiple layouts associated with it. If a small amount of extra information is stored with each assignment, a layout can be chosen based on a secondary optimization criteria such as number of vias or total wirelength.

2.2 First order runtime analysis

The number of possible track assignments for a given column is bounded by $t!$ and the runtimes of all the procedures in next_column() are obviously independent of the channel length n and the total number of signals in the problem instance. (In fact, this global information is not even available within procedure next_column()) For a given channel width t the runtime to process a single

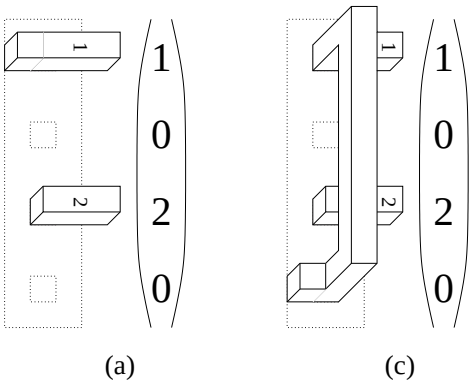

Figure 4: connectPorts() inserts a dogleg in assigment (c) of Figure 3 to connect it to port 1. Assignment (a) is left unchanganged because it alltready connects port 1 correctly. Assignments where a port is blocked by another signal are rejected. (not shown)

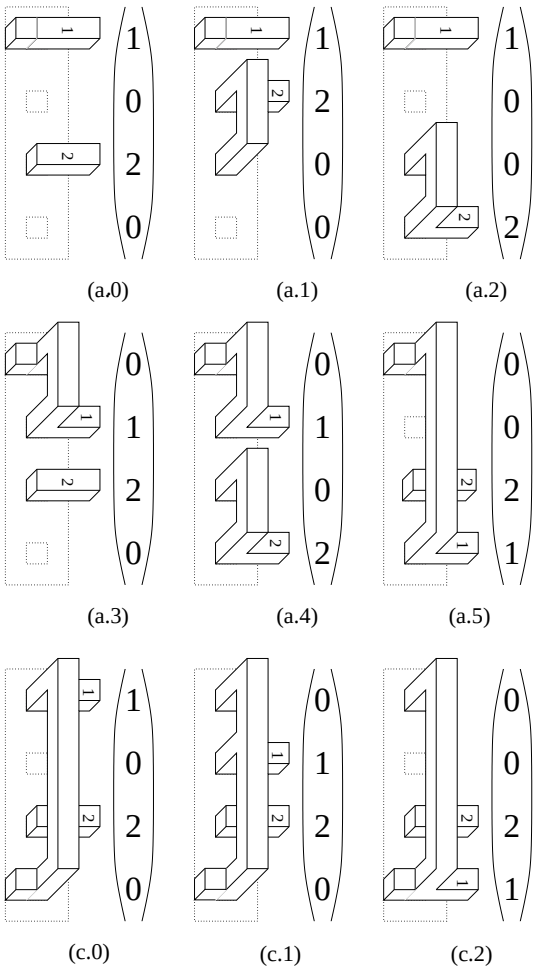

Figure 5: The insertion of all possible combinations of doglegs results in a total of six output track assignments. Note that three of the track assignments can be reached by multiple layouts.

column is therefore bounded by a constant and the overall runtime of the algorithm becomes $O(n)$.

As there are up to $t!$ input assignments to be processed for each column the runtime grows with the channel width t as $\Omega(t^t)$ so that there is no contradiction to the expected exponential complexity due to the NP-completeness of the problem. Algorithms with a behaviour like this are called fixed-parameter tractable problems.

The worst case runtime for each of the assignments processed in a single column is dominated by the dogleg insertion procedure. It must enumerate an exponential number of doglegs. All the other procedures complete in polynomial time. An upper bound for the number of doglegs can be computed as follows:

At a first glance each track in a column can have one of four configurations: A dogleg may or may not connect to the next or previous track, respectively. This immediately leads to an upper bound of 4^t doglegs. This bound can be tightened to 2^{t-1} by the following argument: If we walk down the column track by track we have only two choices for the first track. We either start a dogleg or we do not. In track $1 < i < t$ each of the configurations of tracks 1 to $i-1$ already determines whether there is a dogleg connection between tracks i and $i-1$. The only choice left is to decide whether there is a connection to the next track. This reduces the number of options for each track to 2. Furthermore, there is no choice for the last track, and the total number of doglegs is at most 2^{t-1}.

This bound is not tight, because there are further restrictions on the doglegs. For example, a dogleg beginning on an occupied track can only end on an empty one and vice versa. In most columns the number of possible doglegs is less than t^2.

2.3 Channel width minimization

To find the minimum channel width T the algorithm must be invoked for each channel width, starting with the channel density. The combined runtime is the sum of the runtimes of all invokations.

The runtime of a single invokation grows superexponential with the channel width. The following unequality holds for all functions f that grow at least exponentially: $\sum_{k=1}^{T-1} f(k) < c \cdot f(T)$. The combined runtime of all invokations with a channel width up to T therefore is only by a constant factor larger than if we had known the optimum channel width in advance. The benchmarking results listed in Table 1 show that the overhead for the iterative calling is negligible. The last cell of each table column has a much higer value than all other cells of that column together.

3. IMPLEMENTATION

We implemented three versions of the algorithm. Implementation A uses a hash set to store the set of track assignments. Implementation B uses a heap structure that allows of exact via minimization and some heuristic runtime improvements. Implementation C uses multivalued decision diagrams to represent the set of track assignments by its characteristic function.

In all implementations a track assignment is stored as an array of t integers in the range of 0 to t with the value 0 denoting an empty track and 1 to t denoting one of up to t signals crossing the given column.

3.1 Implementation A

In this implementation the track assignments are stored in a hash set of dynamic size. There are up to $t!$ assignments in the set and the algorithm iterates over the whole set for each of the n columns. The doglegs are generated by a recursive algorithm that needs constant time for each of the $d < 2^{t-1}$ new assignments. The worst case runtime of the algorithm is therefore:

$$\tau_{wc} = O(t!2^t)$$

It is pessimistic to multiply the worst case number of possible doglegs with the worst case number of track assignments as they can never occur simultaneously. For example a column with $t!$ track assignments can have no doglegs at all, and a column with 2^{t-1} doglegs must have less than $t!/(t/2)!$ track assignments.

With this implementation benchmarks with up to 8 tracks could be routed in a reasonable amount of time.

3.2 Implementation B

In this implementation track assignments are stored in a heap using the number of vias needed to create the track assignment as the key. Track assignments with less vias are processed first, resulting in a routing with a minimum number of vias. This produces a circuit that has lower power consumption and smaller delay due to the lower resistance and capacitance of the wires as well as a better yield during manufacturing. Minimizing the number of vias might also result in a smaller layout if compaction is used as a post-processing step.

As a heuristic runtime optimization the assignments were not processed column by column but in a depth-first manner. This is a lot faster in many cases because there usually are a lot of possible optimal routings for a given problem instance and the algorithm can stop as soon as the first one is found instead of enumerating them all.

3.3 Implementation C

In this implementation we use multi-valued decision diagrams (MDDs, [4]) to represent the characteristic function of the set of track assignments. This is an approach similar to the one presented in [9]. But [9]—despite its title—solves the harder maze routing problem. As an effect of the more general approach the worst case runtime there grows exponentially in $t \cdot n$. Our algorithm has a runtime linear in the channel length n.

To store the set of track assignments we construct the characteristic function $f : \{1 \ldots t\}^t \mapsto \{0, 1\}$ that returns 1 for a tupel that is contained in the set of track assignments and 0 for a tupel that is not.

MDD operations can be used to modify all assignments in the set simultaneously. For example to pbtain the subset of all assignments that have an empty track i the cofactor of f for $s_i = 0$ is computed. Some useful operations are reduced to simple pointer redirections.

For an example on how the channel router operates on the MDD set representations, consider column 4 of Figure 1. Figure 5 shows all the valid track assignments that can

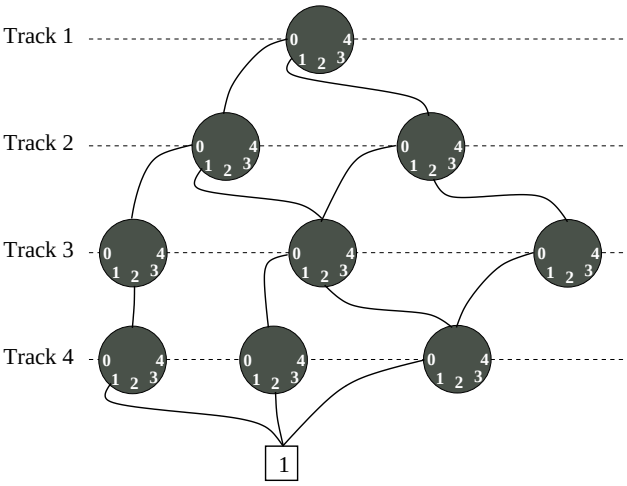

Figure 6: MDD representation for the set of track assignments from Figure 5. There is a path from the root to the 1 terminal for each assignment in the set. Connections to the 0 terminal are omitted for clarity.

enter this column from the left. Figure 6 shows the MDD representation of this set.

Procedure addLeftEdgePorts() must connect the new signal 4 to track 2. This can only be done in track assignments where track 2 is empty. All other track assignments are removed from the set. This is achieved by simply redirecting in level 2 of the mdd the pointers for all values but 0 to the 0 terminal. In our example we, have to change two pointers to remove the three assignments (1, 2, 0, 0), (0, 1, 2, 0) and (0, 1, 0, 2). The resulting BDD is shown in Figure 7. The unused node remains in the data structure for later use.

The insertion of signal 4 in track 2 is also very simple. We just move all the pointers in level 2 that are associated with value 0 to become pointers for value 4. The resulting MDD is shown in Figure 8. Usually the MDD for f will be much smaller than the hash set representation of implementation A. During our benchmarking no MDDs occured that had more than 10^6 nodes even for channels with 13 tracks that might have $13! \approx 6.2 \times 10^9$ distinct track assignments. Also, the MDDs produced by the router are often similar for adjacent tracks and there are some sub-graphs that are used very frequently. The results of operations on these sub-graphs are cached to obtain a substantial speed up. The caching algorithms are build into the MDD package that we used and is transparent to the programmer.

3.4 Experimental Results

Versions A and B of the algorithm where implemented in Java using SUNs JDK 1.1.8. Version C was implemented in C++ using the CU Decision Diagram Package, v2.3 [11].

To obtain a set of channel routing problems for benchmarking we used the PAROS system [10] to perform placement and global routing on a set of benchmarks taken from the ISCAS 85 circuits . The resulting channels where then routed by the three implementations on a Pentium-II processor running at 450 MHz on a Linux system.

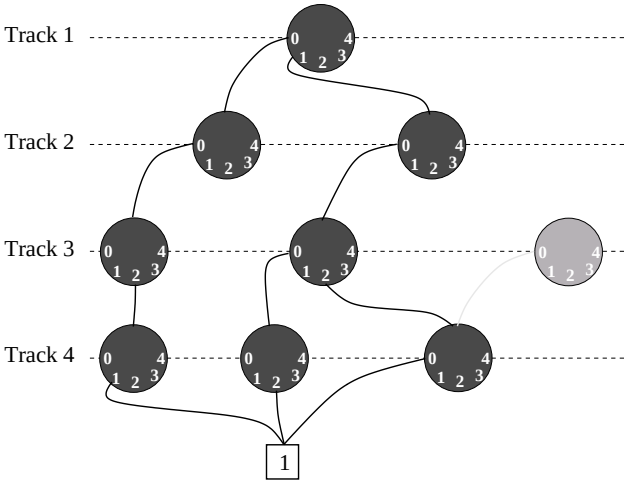

Figure 7: The MDD from Figure 6 after some pointers in level 2 have been changed to remove all track assignments that used track 2.

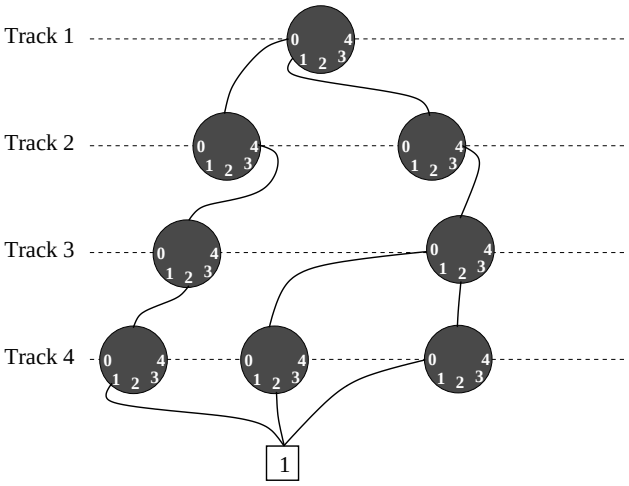

Figure 8: The MDD from Figure 7 after the pointers in level 2 have been moved to indicate that track 2 is occupied by signal 4 in all assignments.

tracks	A	Bmax	Bmin	C
3	0.002	0.004	0.001	0.005
5	0.004	0.005	0.002	0.005
7	0.1	0.008	0.005	0.005
9	>6	5	0.005	0.02
11		>60	0.052	0.3
13				8

Table 1: Runtime per column in seconds for the three implementations of the algorithm

Table 1 lists the runtime per column for each of the algorithms. The runtime of implementations A and C were similar for problem instances with the same channel width. The values shown are the maxima over all instances with that particular width.

The runtime of implementation B however showed significant variation among the benchmarks. The table lists the maximum and minimum runtime for each channel width for implementation B. The great variation in runtime was expected because version B implements heuristics to improve on the runtime. For some benchmarks these heuristics reduce the runtime by a few orders of magnitude and for others they do not succeed at all so that we only see a small increase in the runtime due to the increased algorithmic overhead.

The use of MDDs as a symbolic representation for the set of value assigments improved the runtime for larger channels by several orders of magnitude and allows the routing of channels with about 50% more tracks.

4. CONCLUSION

The results show that the exact solution of channel routing problems can be found in a reasonable amount of time for channels of arbitrary length with up to thirteen tracks using an MDD representation for the set of possible value assignments. The set of operations performed on this data structure probably can be improved to allow for a larger channel width. Additionally, we expect that there exist even better data structures for this algorithm that exploit the fact that there are only $t!$ possible value assignments whereas the MDD structure used can represent sets with t^t elements.

With the current restriction to 13 channels this algorithm can still improve the layout for complex intra-cell routings when it is used in cell generators.

A strength of this algorithm is that it can easily be modified for a large number of different routing styles and secondary optimization criteria as long as the column-by-column principle is preserved. Although we did not test it, we expect that support for single layer doglegs and left to right forks can be added without hurting the runtime too much.

5. REFERENCES

[1] A. Hashimoto and J. Stevens; "Wire Routing by Optimization Channel Assignment within Large Apertures"; *Proceedings of the 8th Design Automation Workshop*, 1971

[2] Ronald L. Rivest and C. Fiduccia; "A greedy channel router"; *Proceedings of the 19th ACM/IEEE Design Automation Conference*; 1982

[3] S. Han and S. Sahni; "Single Row Routing in Narrow Streets"; *IEEE Transactions on Computer-Aided Design*, CAD-3; July 1984

[4] T. Y. K. Kam and R. K. Brayton; "Multi-valued Decision Diagrams"; Technical Report UCB/ERL M90/125, University of California at Berkeley; 1990

[5] Thomas H. Corman, Charles E. Leiserson and Ronald L. Rivest; "Introduction to Algorithms"; ISBN 0-262-53091-0; MIT Press; 1990

[6] Robert L. Maziasz and John P. Hayes; "Layout minimization of CMOS cells"; ISBN 0-7923-9182-9; Kluwer Academic Publishers; 1992

[7] S. Chow, H. Chang, J. Lam and Y. Liao, "The Layout Synthesizer: An Automatic Block Generation System"; *Proc. of CICC;* 1992

[8] Naveed Sherwani; "Algorithms for VLSI Physical Design Automation"; ISBN 0-7923-9592-1; Kluwer Academic Publishers; 1995

[9] F. Schmiedle, Rolf Drechsler and B. Becker; "Exact Channel Routing Using Symbolic Representation"; *IEEE International Symposium on Circuits and Systems (ISCAS)*, Orlando, 1999

[10] Ingmar Neumann, Dominik Stoffel, Hendrik Hartje and Wolfgang Kunz; "Cell Replication and Redundancy Elimination During Placement for Cycle Time Optimization"; *Proc. of the ACM/IEEE Intl. Conference on Computer-Aided Design (ICCAD)*; 1999

[11] Fabio Somenzi; "CUDD: CU Decision Diagram Package"; University of Colorado at Boulder; `http://vlsi.colorado.edu/~fabio/CUDD/`