Report No. 231288

Decision-guided SAT for standard cell routing

Hougardy, Stefan [1]); Schürks, Jan Malte [2])

[1)] Research Institute for Discrete Mathematics and Hausdorff Center for Mathematics, University of Bonn. Email: {hougardy, schuerks}@or.uni-bonn.de

Report No. 231288

Decision-guided SAT for standard cell routing

Hougardy, Stefan; Schürks, Jan Malte

Forschungsinstitut für Diskrete Mathematik
Rheinische Friedrich-Wilhelms-Universität Bonn
Lennéstrasse 2
53113 Bonn
Germany
Telefon +49 228 738770
Telefax +49 228 738771
Electronic Mail: dm@or.uni-bonn.de

Typeset in TEX
Printed at the Rheinische Friedrich-Wilhelms-Universität Bonn

ISSN 1438-9797

Decision-guided SAT for standard cell routing

Stefan Hougardy Jan Malte Schürks
Research Institute for Discrete Mathematics, University of Bonn, Germany
{hougardy,schuerks}@or.uni-bonn.de

October 4, 2023

Abstract

Due to the increasing complexity of the design rules involved, routing standard cells at the transistor level is a challenging problem for traditional routing approaches. Rip-up and reroute algorithms are very fast and compute routings with good netlength, but often fail to find solutions that satisfy all design rules. Integer linear programming based routers can compute optimal solutions but are very slow. SAT-based routers find a solution quickly, but the solution quality may be low. We present a new algorithm that combines the advantages of rip-up and reroute approaches with those of SAT-based routers: Routings are found quickly and are of good quality in practice. At the same time, the algorithm is guaranteed to find a solution whenever one exists and otherwise proves that none exists.

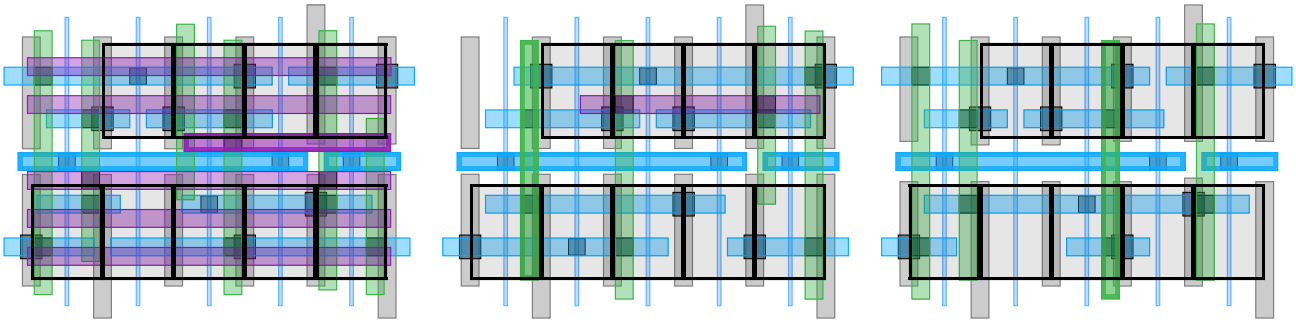
We present experimental results on a cell library of a commercial 4nm node. The 96 cells in this library range in size from an inverter with 2 transistors to a local clock buffer with 243 transistors. For all cells our algorithm finds routings that satisfy *all* design rules. Its runtime is below 11 minutes on all instances. The approach has also been applied in a 3nm setting.

1 Introduction

The increasing complexity of design rules in recent technologies makes it more and more difficult to find routings of standard cells at the transistor level that satisfy all design rules (so called DRC clean routings). In addition a routing needs to optimize certain objectives as for example net length, pin accessibility, or electromigration constraints. Traditional *sequential* routing approaches that route one net after the other fail in this setting even when using sophisticated rip-up and reroute strategies. Instead, in recent years routing approaches that can route all nets *simultaneously* have been applied successfully.

These approaches use a formulation of the routing problem as an integer linear program (ILP) or as a Boolean satisfiability problem (SAT). Both these approaches not only allow to route all nets simultaneously but also provide a rather straightforward way to add new design rules. As powerful solvers for generic ILP [CPL22; Opt23] and SAT [Bie+20] problems exist the main task in designing routing algorithms based on these approaches is to find good formulations of the routing problem as an ILP or as a SAT.

An advantage of formulating a routing problem as an ILP or SAT is that these approaches allow to *prove* that a given instance is *not* routable. By using some more elaborate methods these algorithms may even return the reason why an instance is not routable [XRS02]. However, one should consider that often ILP or SAT formulations are used that artificially limit the routing by precomputing a set of possible Steiner trees for each net [Li+20; RB12], or bound the maximum allowed detour of a net [SR19], or use some restricted routing patterns [NSR99]. In these cases unroutability of an instance is only true with respect to the artificial limitations in the formulation. To really prove



(a) SAT router (used to prove routability) (b) Decision-guided router (c) ILP router (optimal)

Figure 1: Routings of a 2-input XOR gate computed by different routers

that an instance does not have any routing solution one has to use ILP or SAT formulations without limitations as have been used for example in [Cle+20].

In theory a SAT is a special case of an ILP where all variables are allowed to have only value 0 or value 1 and all constraints are of a specific form. Moreover, there exist standard methods to transform any ILP into a SAT [ES06]. Thus, from a theoretical point of view both formulations are equivalent. However, there are some important differences in practice between formulating a routing problem as an ILP or as a SAT. Kang et al. [Kan+18] show that by reformulating an ILP as a SAT in a suitable way one can reduce the runtime to solve a routing problem by several orders of magnitude. Thus, for *deciding* routability a SAT formulation is clearly the better choice. However, an ILP solver can easily not only decide routability but also *optimize* some objective (e.g. net length).

There have been several attempts to include the ability of optimizing some function into the formulation of the routing problem as a SAT. One such approach uses the formulation of counters as Boolean formulas [RB12]. However, in practice counters are limited to count only up to 10 as otherwise the runtime of the SAT solvers explodes. Another approach formulates the routing problem as a MAX-SAT problem [Li+20]. But current MAX-SAT solvers [Bac+22] are significantly slower than SAT solvers.

We will begin by giving a high-level overview of our routing algorithm in section 1.1. This algorithm combines elements of SAT-based routers with elements of sequential routers. We then describe the details of the sequential routing elements used in the algorithm in section 2. Section 3 will give a full description of the SAT-solving element of our algorithm, which will be used in section 4 to improve the parts based on sequential routers. Practical results on 4nm standard cells are presented in section 5.

1.1 Basic approach

We will always assume that the placement of the field effect transistors (FETs) within the cell is fixed. We present a new approach that allows a SAT solver not only to decide routability of such a placement but also to optimize among many feasible routings. For this we extend the approach described in [Nad16]: We begin by converting the ILP from [Cle+20] into a SAT instance, while omitting the multi-commodity flow formulation used to ensure that nets are connected. To solve the resulting SAT instance we will use a modified version of the **conflict driven clause learning (CDCL)** [Bie+21] algorithm, which forms the basis of most modern SAT solvers. The basic idea of CDCL is to successively assign values to variables according to an internal **decision strategy**, applying an algorithm to deduce "obvious" implied assignments after each decision. If at some point all literals of some clause (called the **conflict clause**) are assigned 0, the algorithm uses this clause to "learn" a new clause. It then "jumps

back" to a specific earlier state, where this new clause would have been of use. Refer to chapter 4 of [Bie+21] for a full description of this algorithm. We will extend the algorithm with the following features, which will be formalized in section 3:

1. The internal decision strategy of the solver can be replaced by a problem-specific one.
2. Clauses not implied by the original problem can be added by a **conflict oracle** when they become falsified during the solving process.
3. The backjump distance after a conflict can be extended by problem-specific code.

These features will be used in the following way: The decision strategy will attempt to sequentially build a Steiner tree for each net by marking the corresponding edges as "used". Since the SAT instance encodes all design rules, it is not necessary for these Steiner trees to be DRC clean. When a net becomes unroutable because none of the edges in a cut it has to cross can be used, a clause is added to the instance ensuring that one of the edges is used by the net. The last added feature will be used to avoid "partially routed" nets after backjumping as described in section 4.2. It is worth noting that while the decision strategy is essential for good performance and result quality in practice, it does not affect the correctness of the algorithm. This justifies the term "decision-guided router". Figure 1 shows a comparison of the routing qualities achieved by a SAT solver, an ILP solver, and our new approach.

The decision-guided router can be interpreted as a type of rip-up and reroute algorithm: The decision strategy routes nets while it is possible. When it is no longer possible, the SAT solver will analyze the conflict and backjump accordingly. This corresponds to undoing some of the decisions previously made by the decision strategy, which in turn means ripping out some of the already routed nets.

Unlike [Nad16] we do not only consider the runtime, but also the quality of the resulting routing, and show multiple ways to improve it compared to a basic implementation of the algorithm. We also provide a proof of the correctness of the overall algorithm.

2 Decision-guided routing

2.1 Notation

In the following the nets of a routing instance will be considered as sets of terminals, where each terminal is a subset of the vertices of the routing graph G. A routing solution for a net N is a connected subgraph R of G such that $V(R) \cap T \neq \emptyset$ for all terminals $T \in N$. For simplicity we will assume that the terminals of a net are pairwise disjoint. For a net N we say that a cut $\delta(C) := \{e \in E(G) : |e \cap C| = 1\}$ with $C \subseteq V(G)$ **separates** N if there are terminals $T_1, T_2 \in N$ such that $T_1 \subseteq C$ and $T_2 \subseteq V(G) \backslash C$. Let H be a subgraph of G with $V(G) = V(H)$. Clearly a routing solution for N exists in $E(H)$ if and only if there is no cut $\delta(C)$ in G separating N with $\delta(C) \cap E(H) = \emptyset$. While a routing solution is not necessarily a tree, it always contains a Steiner tree (formally a group Steiner tree) on the terminals of N. So we will compute Steiner trees for each net, which may be proper subsets of the final routing solution of the net.

In addition to the routing-related notation above, we will need some SAT-related notation to describe the decision strategy and conflict oracle used in routing: Let V be a set of binary variables. A literal is either a variable x or its negation $\overline{x}$ for $x \in V$. A clause is a set of literals.

We call a function $f : V \to \{0, 1, \varepsilon\}$ a **partial assignment**, where the value ε denotes that no "real" value is assigned to the variable. We denote by $\text{Undef}(f) := f^{-1}(\varepsilon)$ the set of variables left unassigned by f. For $x \in \text{Undef}(f)$ we define $f(\overline{x}) = \varepsilon$. For $x \in V \setminus \text{Undef}(f)$ we define $f(\overline{x}) = 1 - f(x)$. A **full assignment** is a partial assignment f with $\text{Undef}(f) = \emptyset$. For a partial assignment f, $v \in \text{Undef}(f)$ and $x \in \{0, 1\}$ we define the partial assignment $f \cup (v, x)$ to map v to x and all $v' \neq v$ to $f(v')$.

2.2 The decision strategy

We denote by u_e ($e \in E(G)$) and u_v ($v \in V(G)$) the indicator variables deciding whether an edge respectively a vertex is used in a full solution. For $N \in \mathcal{N}$ the variables u_e^N ($e \in E(G)$) and u_v^N ($v \in V(G)$) decide whether e respectively v is used by net N. Clearly u_e^N (u_v^N) implies u_e (u_v), and at most one of the variables u_e^N resp. u_v^N for a fixed edge e resp. a fixed vertex v can be active in a satisfying solution. Additionally $u_{\{v,w\}}^N$ for $\{v, w\} \in E(G)$ implies u_v^N and u_w^N.

The conflict oracle will not be given as a full algorithm here. It has to check if each net N is connected within G_N^ε. If it is not, it computes a cut $\delta(C)$ separating terminals of N with $\delta(C) \cap E(G_N^\varepsilon) = \emptyset$ and then returns a clause $\bigvee_{e \in \delta(C)} u_e^N$. In practice this step is part of the Steiner tree computation in the decision strategy, but is always run on all nets even if an earlier net returns a decision literal. The basic structure of the decision strategy for cell routing is given in algorithm 1, which is very similar to the heuristic used in [Nad16].

Input: Partial assignment f; routing graph G; list of nets $\mathcal{N}$
Output: Either a decision literal in Undef(f) or the message "Use solver heuristic"
1 **foreach** *net* $N \in \mathcal{N}$ **do**
2 Compute $G_N^1 := (V(G), \{e \in E(G) : f(u_e^N) = 1\})$
3 Compute $G_N^\varepsilon := (V(G), \{e \in E(G) : f(u_e^N) \neq 0\})$
4 **if** *the terminals of* N *are not connected in* G_N^1 **then**
5 Compute a Steiner tree T for N in G_N^ε
6 Choose $e \in T$ with $u_e^N \in$ Undef(f)
7 **return** decision u_e^N
8 **end**
9 **end**
10 **if** *there is* $v \in V(G)$ *with* $u_v \in$ Undef(f) **then**
11 **return** decision $\overline{u_v}$
12 **end**
13 **return** "Use solver heuristic"

Algorithm 1: Basic decision strategy

Since the internal decision heuristic of the SAT solver tends to generate much floating metal, the "undecided" vertices are greedily disabled after all nets have been routed. As described before the structure of the overall algorithm mirrors that of a rip-up and reroute search: If some net cannot be routed due to a conflict with some existing net, some of the existing nets are removed and the routing is repeated under different constraints and/or a different net order (see section 2.3). However, unlike in a usual rip-up and reroute approach, this step is not performed by a heuristic, but by the SAT solver. Because of this, the algorithm guarantees that a solution will be found if one exists and can be used in settings like cell routing, where traditional rip-up and reroute frequently fails due to complex design rules. Additionally the SAT solver handles all design rules, so the Steiner tree algorithm does not need to be modified for each new type of design rule.

2.3 Ordering the nets

Clearly the behavior of the router heavily depends on the order the nets are checked by the loop in line 1. Initially the nets are ordered by increasing minimum distance to the vertical cell boundaries. The reasoning behind this ordering is that special rules apply at the cell boundaries to ensure that arbitrary cells can be placed directly adjacent to each other. In the current technology, some of these *boundary conditions* differ between tracks. In some

cases this means that only specific tracks can be used to connect to terminals very close to the cell boundaries. Routing these nets late frequently leads to convoluted routings: The tracks which can be used at the boundaries may be available there, but unavailable further inside the cell. The net can often still be routed in this case, but requires additional track changes when it could have been routed on a single track otherwise.

Afterwards the net order is changed every time the conflict oracle returns a cut separating a net N: this net is moved to the front of $\mathcal{N}$. The algorithm would be correct even when this is not done, however even a simple net conflict would only be resolved after adding a large number of cut clauses. Instead the net which could not be routed previously is routed before the net that prevented it from being routed, again mirroring the behavior of rip-up and reroute routers.

2.4 Finding Steiner trees

The basic structure of the algorithm for finding the per-net Steiner trees in line 5 is given in algorithm 2. It is based on a Prim-like heuristic: A single terminal is chosen as the start terminal. Then, while some terminal is not connected to the current tree, we run Dijkstra's algorithm starting from all vertices in the current tree. The first new terminal reached by Dijkstra's algorithm (i.e. the one closest to the current tree) is then connected to the tree by a shortest path. During this path search, edges that are already used by the net (i.e. edges with $f(u_e^N) = 1$) are given cost 0.

Input: Net N, routing graph G, partial SAT solution f
Output: Steiner tree for N or a cut $\delta(C)$ separating N s.t. $f(e) = 0$ for all $e \in \delta(C)$
1 Choose starting terminal $t \in N$ and set $D \leftarrow \{t\}$
2 $T \leftarrow (t, \emptyset)$
3 Define $c'(e) = \begin{cases} c(e) & \text{if } f(u_e) = \varepsilon \\ 0 & \text{if } f(u_e) = 1 \\ \infty & \text{if } f(u_e) = 0 \end{cases}$
4 **while** $D \neq N$ **do**
5 | Compute a partial shortest path tree S in $G/V(T)$:
6 | | • Start vertex: $V(T)$
7 | | • Cost function: c'
8 | | • Consider edges with cost ∞ to be absent
9 | | • Stop if a terminal in $N \backslash T$ is reached
10 | **if** *no terminal in* $N \backslash T$ *is reached in* S **then**
11 | | **return** the cut $\delta(V(S))$
12 | **end**
13 | Let t' be a terminal in $N \backslash T$ reached by S
14 | Let P be the path in S from $V(T)$ to a vertex of t'
15 | **if** *this is the first iteration* **then**
16 | | $T \leftarrow P$
17 | **else**
18 | | $T \leftarrow (V(T) \cup V(P), E(T) \cup E(P))$
19 | **end**
20 **end**
21 **return** T

Algorithm 2: Basic Steiner tree algorithm

Since it would be very slow to recompute Steiner trees every time a decision or conflict is queried, the computed tree is cached until some edge of it becomes unusable. In addition to the immediate runtime improvement this also "stabilizes" the algorithm in the sense that a net will not change its routing unless it becomes necessary, even across backjumps. On the other hand this introduces an additional source of suboptimality, since the routing may no longer be optimal after backjumps.

2.5 Improving tree quality

By itself algorithm 2 is a 2-approximation for the Steiner tree problem, and usually yields good trees in practice. However, the practical quality can be improved in a number of ways without major runtime impacts:

If a net acts as a logical input or output of the final cell, it needs to provide a *pin* that external routing can be connected to. This is typically modeled by adding a terminal which consists of all vertices on the lowest metal layer. If the Steiner tree algorithm is implemented as described, this will be one of the first terminals to be connected as it can easily be reached from all terminals. However, it will be connected by adding metal on an essentially random track. In many cases this is not useful for connecting to the remaining terminals and may even hinder those connections due to design rules. Additionally the net will end up blocking multiple tracks when a single one would have been enough. To avoid this issue, terminals consisting of vertices on metal layers (as opposed to FET vertices) are only considered valid "targets" after all FET terminals have been connected. In many cases, connecting all FETs requires the use of the lowest metal layer (M1) anyway, so the pin terminal is automatically connected.

Another cause of bad trees is the choice of the path P in line 14: If the path is chosen to be close to the remaining terminals, there is likely to be a way to connect them to some internal vertex of the path. Otherwise it may be necessary to connect them to the endpoints at higher cost. In practice this is implemented by ordering the labels of equal weight in Dijkstra's algorithm which correspond to paths reaching a previously unconnected terminal. These are compared according to the sum of the minimum distances of the path to the unconnected terminals. This particularly improves routing quality for nets which have terminals in both the N- and P-region of a circuit row on some tracks and only in one of these regions on other tracks. Connecting terminals of the first type by a badly chosen M1 track might require additional tracks to connect to the terminals of the second type, while now a track will be chosen which can connect to all terminals.

The choice of the starting terminal t in line 1 offers another chance to obtain better Steiner trees. In many cases, some FET terminal is in a position where only one or two tracks on M1 can be used to connect to it due to existing wiring. If only a single terminal t_0 is in this situation, choosing some other terminal as the starting terminal may result in decreased routing quality: The M1 track used to connect these terminals can be chosen arbitrarily. Since the algorithm is not aware of t_0 yet, the track will be chosen without considering its restrictions and typically cannot be used to connect to t_0. To connect this partial routing to t_0, it is necessary to use higher metal layers, which are also used in inter-cell routing and thus should be kept as empty as possible. By choosing t_0 as the terminal with the fewest usable M1 tracks, usage of higher layers can be avoided in some cases. For similar reasons the cost of M1 edges are decreased for tracks which can likely be used to connect large portions of the net.

3 Decision-guided SAT solving

Boolean constraint propagation (BCP) is an important subroutine of CDCL which is used to infer additional assignments implied by a decision: If a clause exists in which all literals but one are assigned 0, the remaining literal has to be assigned 1 to satisfy the clause. This step is repeated while such a clause exists. If a clause is found in which all literals are assigned 0, this clause is a conflict clause and is returned instead of the extended assignment. We say that a partial assignment f is **fully propagated** under a set of clauses $\mathcal{C}$ if BCP applied to f and $\mathcal{C}$ does not deduce any new variable assignments. Otherwise we say that f **propagates under** $\mathcal{C}$.

We now formally define the conflict- and backjump oracle described in 1.1:

Definition 1. *Let V be a set of Boolean variables.*

*A **conflict oracle** for a set $\mathcal{D}$ of clauses on V is a function c such that, given a partial assignment f on V, $c(f)$ is some clause in $\mathcal{D}$ which is falsified by f if such a clause exists. Otherwise $c(f) := \varepsilon$.*

*A **backjump oracle** on V is a function mapping a stack of partial assignments on V to $\{0, 1\}$.*

Formally speaking, the main use of a conflict oracle is to represent an exponential-size clause set in a compact form. Especially with a good decision strategy the CDCL algorithm will typically only query a very small fraction of these clauses.

The modified CDCL algorithm uses a function analyzeConflict. This function does not change compared to a standard CDCL algorithm, so we only describe its properties rather than its implementation. Let C_0 be a clause falsified by the current assignment f in the algorithm, but not by $\text{top}(T)$ if T is nonempty. Then we require that the following properties are fulfilled for the clause $C_1 :=$ analyzeConflict(C_0):

1. C_1 can be obtained by applying resolution to C_0 and elements of $\mathcal{C}$, i.e. deducing $A \vee B$ from two clauses $A \vee x$ and $B \vee \overline{x}$.
2. C_1 is falsified by f.
3. If T is empty, $C_1 = \emptyset$. Otherwise $\text{top}(T)$ is not fully propagated under $\{C_1\}$, but does not falsify it.

The full details of the analyzeConflict function can be found in chapter 4 of [Bie+21].

Theorem 1. *"CDCL for decision-guided algorithms" (algorithm 3) fulfills the following properties:*

1. *If an assignment is returned, the assignment satisfies $\mathcal{C} \cup \mathcal{D}$.*
2. *If UNSAT is returned, no assignment satisfying $\mathcal{C} \cup \mathcal{D}$ exists.*
3. *The algorithm terminates in finite time.*

Proof. If f becomes a full assignment in line 22, it satisfies $\mathcal{C}$ (which is a superset of the original instance) since it is a full assignment returned by BCP, and $\mathcal{D}$ since c did not return a conflict clause. Now assume that f becomes a full assignment in line 30. For this to happen, f must have been fully propagated under $\mathcal{C}$ at the start of the iteration and must assign all variables except for one variable $x_0 \in V$. So for every clause $C \in \mathcal{C}$, we have either $|\{c \in C | f(c) \neq 0\}| > 1$ or $1 \in f(C)$. Since x_0 and $\overline{x_0}$ are the only literals mapped to ε, the first condition implies either $\{x_0, \overline{x_0}\} \subseteq C$ or $f(c) = 1$ for some $c \in C$. In all cases the clause is satisfied independent of the value of x_0, so the assignment after line 30 satisfies $\mathcal{C}$. Line 30 will never be reached if $f \cup (v, x)$ falsifies a clause in $\mathcal{D}$ (line 27). Hence if an assignment is returned, it satisfies $\mathcal{C} \cup \mathcal{D}$.

When handleConflict is called, C_0 is always in $\mathcal{C} \cup \mathcal{D}$ since it is returned by either BCP on $\mathcal{C}$ or the conflict oracle for $\mathcal{D}$. No clause in $\mathcal{C} \cup \mathcal{D}$ is falsified by $\text{top}(T)$, so the call to analyzeConflict fulfills the properties required above. As the clauses returned by analyzeConflict are obtained by resolution steps from clauses of $\mathcal{C}$ and C_0, adding them to $\mathcal{C}$ does not affect the satisfiability of the instance $\mathcal{C} \cup \mathcal{D}$. So if UNSAT is returned, adding the empty clause (which is never satisfied) to $\mathcal{C} \cup \mathcal{D}$ would not change satisfiability, so the original instance must have been unsatisfiable.

At the end of each iteration of the main loop where T is not empty, f is an extension of $\text{top}(T)$: At the start of the algorithm, T is empty. In each iteration, there are three possibilities: f is extended by unit propagation, a new variable is assigned, or a conflict is found and f is replaced by some element from T. In the first case T does not change, so the new value of f, which extends the old value of f, still extends $\text{top}(T)$. In the second case, $\text{top}(T)$

Input: A SAT instance $\mathcal{C}$ with variable set V, a clause set $\mathcal{D}$ given by a conflict oracle c, a backjump oracle b

Output: A satisfying assignment f for $\mathcal{C}$ and $\mathcal{D}$ or the message UNSAT

Let f be the empty partial assignment on V
Let $T = ()$ be a stack of partial assignments

Function *handleConflict(conflict clause C_0)*
    $C_1 \leftarrow$ analyzeConflict(C_0)
    if $C_1 = \emptyset$ **then**
        Halt and return UNSAT from the entire algorithm
    end
    Add C_1 to $\mathcal{C}$
    while $T \neq ()$ *and* top(T) *is not fully propagated under* $\mathcal{C}$ **do**
        $f \leftarrow$ top(T), and remove top(T) from T
    end
    while $T \neq ()$ *and* $b(T) = 1$ **do**
        $f \leftarrow$ top(T), and remove top(T) from T
    end
end

while f *is not a full assignment* **do**
    if f *is not fully propagated under* $\mathcal{C}$ **then**
        Apply Boolean constraint propagation to f
        if *either BCP or* $c(f)$ *returns a conflict clause* C_0 **then**
            handleConflict(C_0)
        else
            Replace f by the assignment returned by BCP
        end
    else
        Push f onto T
        Choose a variable $v \in$ Undef(f) and $x \in \{0, 1\}$
        if $c(f \cup (v, x))$ *returns a conflict clause* C_0 **then**
            handleConflict(C_0)
        else
            $f \leftarrow f \cup (v, x)$
        end
    end
end
return f

Algorithm 3: CDCL for decision-guided algorithms

at the end of the iteration will be the value of f before the iteration, which is extended by the new value of f. In particular, since this is the only case where an element is pushed to T, "deeper" elements of T are always extended by elements "further up" in the stack. In the last case f at the end of the iteration is an element of T at the start of the iteration, while top(T) (if it exists) is the element which was directly below the one now assigned to f. So f extends top(T) in all cases.

At the end of each iteration of the loop, all elements of T are fully propagated under the current set $\mathcal{C}$: At the start of the first iteration T is empty and the statement is trivial. If a new assignment is pushed onto T (line 25), it was fully propagated. If $\mathcal{C}$ is modified (line 8), elements are popped from T until the condition is satisfied for top(T). Since the elements were fully propagated at the start of the iteration, we only need to show that no element of T propagates under the added clause C_1. If T is empty, this is trivial. Otherwise the third property of analyzeConflict guarantees that top(T) (in line 4) propagates under C_1. top(T) after line 11 is extended by this assignment and does not propagate under C_1, so at least two literals of C_1 are unassigned. Since all other elements of T are extended by top(T), they also leave at least two literals of C_1 unassigned and thus are fully propagated under C_1 and the new value $\mathcal{C}$. Popping further elements from T does not affect this.

To see that the algorithm terminates, observe that in line 8 C_1 was not already present in $\mathcal{C}$: If line 8 is reached, T is not empty as otherwise analyzeConflict would have returned an empty clause and the algorithm would have terminated in line 6. So top(f) exists and propagates under $\{C_1\}$ but does not propagate under $\mathcal{C}$, which implies $C_1 \notin \mathcal{C}$. So any time handleConflict is executed, $|\mathcal{C}|$ is increased by 1. Since there are only $3^{|V|}$ possible clauses (each variable can be contained positively, negatively, or not at all), this can only happen a finite number of times. Any iteration of the loop which does not enter analyzeConflict extends f by at least one assigned variable (either by a decision or by unit propagation), so in every $|V|$ iterations of the outer loop either a conflict clause will be found or a full assignment is returned. This limits the number of executions of the loop to $3^{|V|} \cdot |V|$. □

4 SAT-based router improvements

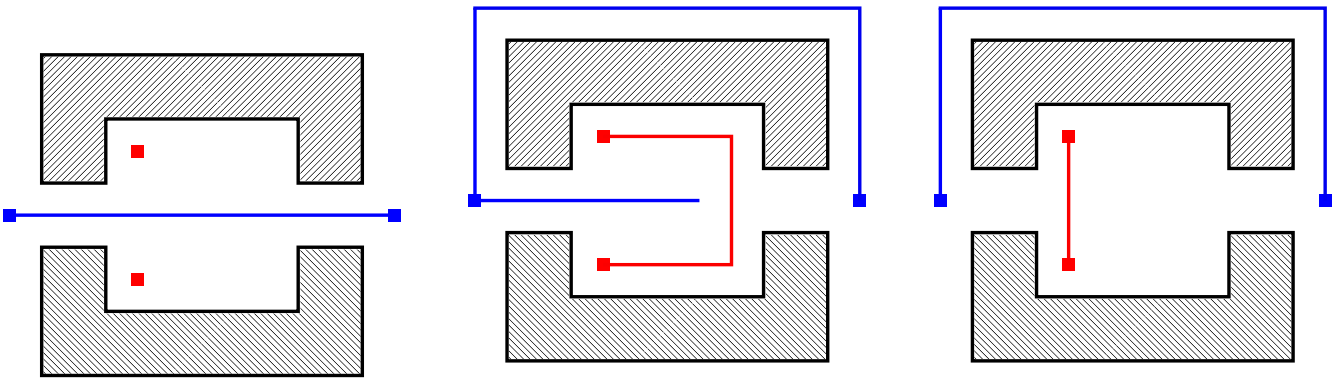
Using the details of the SAT solving algorithm, there are two further improvements that can be made to the routing algorithm described in section 2.

4.1 Finding legal trees

In many cases the paths found in algorithm 2 violate design rules even though legal paths of similar length exist. This leads to higher runtime, but also degrades routing quality since it is often possible to extend a failed routing to a full routing at much higher cost. In typical routers this is addressed by making the path search aware of specific design rules (see e.g. [Ahr20]). However, this approach can only consider relatively basic rules. Additionally it would introduce a high dependency on the design rules of the current technology into the Steiner tree code, which we would like to avoid.

Instead, it is possible to apply BCP to partial routings for a net during the Dijkstra path search to detect at least basic DRC violations: Denote the current partial assignment by f and let g denote the assignment resulting from a partial routing of a net. If BCP applied to $f \cup g$ and the current $\mathcal{C}$ in algorithm 3 yields a conflict, the routing is not a valid extension of f. By running such a BCP check on every label created during the Dijkstra path search and considering labels where BCP failed to be strictly longer than any label where BCP succeeded, legal paths can be found in many cases where a simple Dijkstra would have failed to find one. However, this is not exact: the approach may fail to find a legal path even though one exists, so labels for which BCP failed still have to be considered.

Unfortunately this approach is quite slow in practice: If BCP is run on all labels and considers all clauses, the runtime is dominated by this check and much higher than it would be otherwise. An obvious optimization



(a) Conflict after routing the blue net

(b) Solution after standard backjumping

(c) Solution after extended backjumping

Figure 2: Standard backjumping can result in suboptimal routings

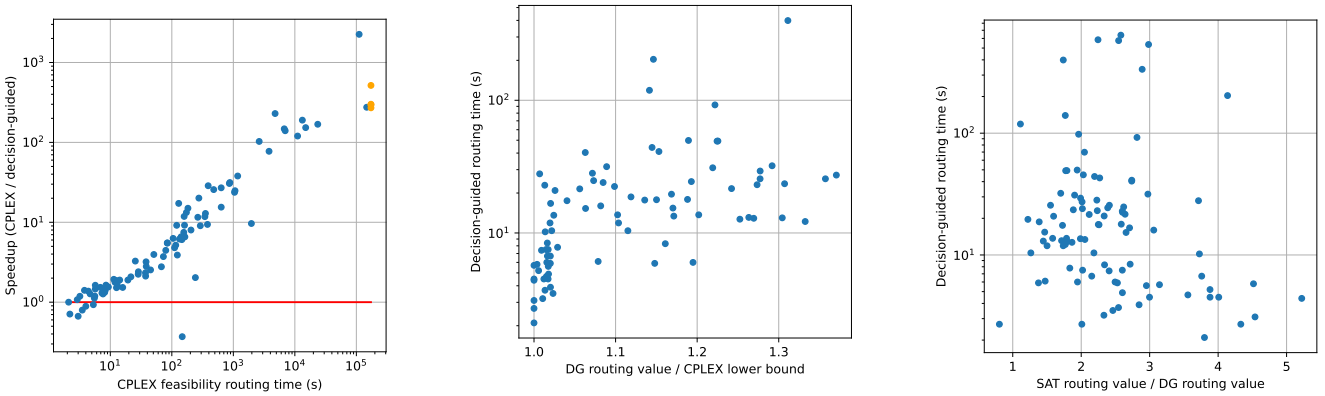
is to compute each path without BCP first, and re-run Dijkstra's algorithm with BCP only if applying BCP to the resulting path yields a conflict. If only binary clauses (i.e. clauses containing only two literals) are considered, the runtime impact can be reduced, but fewer violations can be detected: These clauses correspond to simple implications, and the input is stable under unit propagation. So it is sufficient to detect if there is a variable x for which both x and $\overline{x}$ are implied by the partial routing. An alternative approach is to only apply BCP once routing a net without BCP failed repeatedly. Lastly BCP might only be applied after traversing an edge "likely to cause conflicts", for example vias. In practice all of these options turned out to either not improve routing quality significantly except on very few cells, or unacceptably increased runtime. Despite of this, it still seems like it should be possible to use BCP in some way to avoid at least trivial design rule violations.

4.2 Extended backjumping

Without a custom backjump oracle b, algorithm 3 only undoes decisions (i.e. edge assignments) while the learned clause stays asserting, i.e. while it will yield a new assignment using BCP. In the example in figure 2 this leads to suboptimal routing of both nets: The blue net is routed first, assigning the edges of the shortest path to 1 from left to right as in figure 2a. This introduces a conflict in the red net, since it has to use one of the vertical edges in the open area in the middle of the blockage (hatched area). With the default backjump strategy partial assignments will be removed from T until more than one of the edges becomes usable in top(T). After this, the red net can be routed, but with a significant detour. The blue net can also still be routed, but still uses edges of the original routing that are not necessary in the new routing. The resulting routing is shown in figure 2b. Similar issues can even occur within a single net, when part of the old routing is in conflict with the new routing due to design rules.

To address this issue, the backjump oracle can be used to ensure that either no decisions or all decisions of a net are removed during backjumping. This results in the routing shown in figure 2c. However, in some cases this can drastically increase the number of conflicts required to legally route a net. To avoid this, the algorithm falls back to partially removing a net after removing it completely 16 times in succession[1]. Specifically, each element of T is annotated with the number of times it became top(T) after lines $8-11$ of algorithm 3. Before entering lines $10-11$, the element T that would become top(T) is computed. If the annotation is 16 or more, the loop is skipped.

[1]The limit was chosen arbitrarily, but appears to work well in practice.



(a) Speedup of the decision-guided router compared to CPLEX configured for feasibility

(b) Factor between the lower bound on netlength computed by CPLEX and the netlength achieved by decision-guided routing

(c) Factor between the netlengths achieved by SAT routing and decision-guided routing

Figure 3: Comparison between ILP routing and decision-guided routing

5 Experimental results

The modifications to the CDCL algorithm in algorithm 3 are not part of common solver implementations. Additionally, due to the complexity of the solvers and the nontrivial interactions between various features, it would not be easy to add these modifications to highly sophisticated solvers like CaDiCaL [Bie+20]. However, the performance of the actual SAT solver is not overly critical for this algorithm: no large instance spends significantly more than 50% of its runtime in the SAT solver, some even spend less than 10% there. Therefore the modifications were implemented as part of a custom basic CDCL implementation. For decision-guided routing the restart strategy is set to always restart after 256 conflicts, rather than increasing this limit over time. This results both in faster runtimes and better results, since the router does not spend as much effort on attempting to complete a partial routing which can either barely be extended to a full routing or barely cannot be extended. Such routings "near the edge" of routability are generally very convoluted and therefore not the type of routing we want to compute unless no others exist.

Since routing is split into multiple phases in our tool, FETs are forced to connect to the first metal layer not only by the multi-commodity flow formulation but also by direct constraints. These constraints were disabled during the initial development of the decision-guided router. This was a simple measure to ensure that cells were not routed correctly "by accident", so that it was easier to see if the decision-guided router was actively connecting them. For the tests below these constraints are included in the clause set $\mathcal{C}$ again: Many cells contain "congested" regions where many different nets need to be connected to FETs in a small area. Including all constraints forcing FET connections to M1 upfront allows the SAT solver to reason about this requirement without having to run into conflicts for every clause.

The test runs were performed with up to 124 runs in parallel on a dual AMD EPYC 7742 machine with 128 CPUs and 2048 GB main memory. Each run used a single thread with a time limit of 48 hours. The testbed consists of FET placements for 96 cells. 45 of these are logic cells, 36 are latches, 6 are LCBs (local clock buffers) with 118-243 FETs and 46-89 nets, and 9 are subcells of LCBs. 18 of these cells are multi-row cells, with the largest placement spanning 59 gate pitches and 4 circuit rows. The design rules and cell structures are those of a

commercial 4nm technology node. Similar results have been obtained in a 3nm context. We use CPLEX to solve the ILP directly. It is configured for pure feasibility for the runtime comparison and utilizes a routing corridor heuristic to improve performance. The decision-guided router incorporates all optimizations discussed above except (as noted before) applying BCP during path search since this led to increased runtimes without improved routing quality on many cells.

Figure 3a shows the speedup of the decision-guided router compared to the runtime of CPLEX. Instances where CPLEX failed to find a solution are shown in orange. The decision-guided router was able to find routings for all cells within slightly above 10 minutes, while CPLEX failed to find routings on some of the larger cells even after 48 hours of runtime. The speedup factor appears to grow polynomially in the CPLEX runtime: The speedup exceeds 100 for virtually all instances with a CPLEX runtime of over an hour, and even exceeds 1000 on one instance. The memory usage stays below 15 GB on all instances.

Figure 3b shows the quality of the routings found by the decision-guided router. The lower bounds on the required routing netlength were obtained using CPLEX, configured for 2% optimality gap (difference between best known feasible solution and best lower bound) with a time limit of 48 hours. To speed up the computation the solution found by the decision-guided router was passed to CPLEX as a starting solution. The horizontal position of each point gives the ratio between the objective value attained by the routing found by the decision-guided router and the lower bound on the optimum routing value computed by CPLEX. On very large instances CPLEX did not reach a reasonably small optimality gap, so the ratio of the objective value attained by the decision-guided router with the lower bound is much larger than that with the optimum objective value. Therefore instances are excluded from the plot if the optimality gap is at least 10% and the integral solution found by CPLEX is identical to the one found by decision-guided routing (i.e. the starting solution). The vertical position of the points indicates the runtime of the decision-guided router on the corresponding cell.

Two additional cells are omitted from figure 3b: a very small 2-input NAND and the corresponding NOR gate. On these cells the solutions computed by the decision-guided router differ from the optimal solution by a factor 1.58 (NOR) and 3.85 (NAND) respectively, much higher factors than on any other cells. Including these outliers in the plot would have drastically reduced readability, especially since the performance on small cells is not critical.

If these outliers are ignored, the routing quality is very reasonable for such a fast router: All routings where this can be determined are within 40% of the optimal netlength, and only few are more than 30% longer than the optimum. Especially on many smaller cells the routing is even optimal or close to it. While the routings are not as good as those computed by the ILP solver, they are an extremely useful tool during manual modification of a layout, where they can be used to quickly assess the routability impact of placement changes.

Figure 3c demonstrates the improvement in routing quality compared to a plain SAT router: On most instances the solution of the SAT router is more than twice as long as that computed by the decision-guided router. The routing runtimes are very similar between the two approaches on reasonably small instances. However, the decision-guided router is significantly faster on very large instances.

6 Future work

While our results are already useful for many practical purposes, it seems likely that the routing quality can be improved further without a major increase in runtime. The most promising ideas for this are given below:

- Recent tests indicate that the main source of suboptimality in the decision-guided router is the choice of the tracks used to access FETs: If this choice is fixed to the one used by the decision-guided router, the ILP solver cannot improve the routing significantly on most instances. Therefore an obvious way to improve routing quality is to compute preferred tracks for each FET contact "globally" instead of leaving the choice to the Steiner tree computation of individual nets.

- In some cases long routings of a net result from the start of a short routing which was illegal, but whose start could be extended to a longer legal routing. In these situation it might be useful for the router to "ask" the SAT solver to remove the entire net as in section 4.2 and then put more effort into computing a legal routing as in section 4.1. The number of these "user-requested" backjumps would have to be limited to avoid infinite cycling.

References

[Ahr20] Markus Ahrens. "Efficient Algorithms for Routing a Net Subject to VLSI Design Rules". Dissertation. Forschungsinsitut für Diskrete Mathematik, Universität Bonn, 2020.

[Bac+22] Fahiem Bacchus, Jeremias Berg, Matti Järvisalo, Ruben Martins, and Andreas Niskanen. "MaxSAT Evaluation 2022 : Solver and Benchmark Descriptions". In: *Department of Computer Science Series of Publications B* (2022). URL: https://helda.helsinki.fi/handle/10138/347396.

[Bie+20] Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximilian Heisinger. "CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling Entering the SAT Competition 2020". In: *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*. 2020, pp. 50–53.

[Bie+21] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh. *Handbook of Satisfiability*. IOS Press, 2021. ISBN: 978-1-64368-160-3.

[Cle+20] Pascal Van Cleeff, Stefan Hougardy, Jannik Silvanus, and Tobias Werner. "BonnCell: Automatic Cell Layout in the 7-nm Era". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39 (2020), pp. 2872–2885.

[CPL22] CPLEX. *IBM ILOG CPLEX Optimization Studio 22.1.1*. 2022. URL: https://www.ibm.com/docs/en/icos/22.1.1.

[ES06] Niklas Eén and Niklas Sörensson. "Translating Pseudo-Boolean Constraints into SAT". In: *Journal on Satisfiability, Boolean Modeling and Computation* (2006), pp. 1–26.

[Kan+18] Ilgweon Kang, Dongwon Park, Changho Han, and Chung-Kuan Cheng. "Fast and Precise Routability Analysis with Conditional Design Rules". In: *SLIP '18: Proceedings of the 20th System Level Interconnect Prediction Workshop*. 2018, Article No.: 4, Pages 1–8. DOI: 10.1145/3225209.3225210.

[Li+20] Yih-Lang Li, Shih-Ting Lin, Shinichi Nishizawa, and Hidetoshi Onodera. "MCELL: Multi-Row Cell Layout Synthesis with Resource Constrained MAX-SAT Based Detailed Routing". In: *Proceedings of International Conference on Computer Aided Design*. 2020, 12A.2.

[Nad16] Alexander Nadel. "Routing under constraints". In: *2016 Formal Methods in Computer-Aided Design (FMCAD)*. 2016, pp. 125–132. DOI: 10.1109/FMCAD.2016.7886670.

[NSR99] Gi-Joon Nam, Karem A. Sakallah, and Rob A. Rutenbar. "Satisfiability-Based Layout Revisited: Detailed Routing of Complex FPGAs Via Search-Based Boolean SAT". In: *FPGA 99, Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*. 1999, pp. 167–175.

[Opt23] Gurobi Optimizer. *Gurobi Optimizer Reference Manual 10.0*. 2023. URL: https://www.gurobi.com/documentation/current/refman/index.html.

[RB12] Nikolai Ryzhenko and Steven Burns. "Standard Cell Routing via Boolean Satisfiability". In: *DAC 2012, Proceedings of the 49th Annual Design Automation Conference*. 2012, pp. 603–612.

[SR19] Anton Sorokin and Nikolay Ryzhenko. “SAT-Based Placement Adjustment of FinFETs inside Unroutable Standard Cells Targeting Feasible DRC-Clean Routing”. In: *GLSVLSI'19, Proceedings of the 2019 Great Lakes Symposium on VLSI*. 2019, pp. 159–164.

[XRS02] Hui Xu, Rob A. Rutenbar, and Karem Sakallah. “sub-SAT: A Formulation for Relaxed Boolean Satisfiability with Applications in Routing”. In: *ISPD'02, Proceedings of the 2002 International Symposium on Physical Design*. 2002, pp. 182–187.