Routing under Constraints

Alexander Nadel
Intel Corporation, P.O. Box 1659, Haifa 31015 Israel
Email: alexander.nadel@intel.com

***Abstract*—Routing is an essential stage in physical design, where already placed components are connected by wires. Routing must satisfy various manufacturing requirements, referred to as design rules. We formalize the problem of design-rule-aware routing and introduce a solver, called `DRouter`, for the resulting problem. Plain routing is often modeled as follows: given an undirected weighted graph and a set of m disjoint nets (each net being a set of vertices), a routing is a (minimal) forest of m disjoint trees, where each tree spans a net. `DRouter`'s input comprises a plain routing instance and a bit-vector formula, whose variables include the edges of the graph as Boolean variables (along with other variables). `DRouter` looks for a satisfying assignment to F, such that the satisfied edges comprise a routing. `DRouter` implements an A*-based router inside a SAT solver. It overrides the solver's decision and restart strategies and enhances its learning with routing-aware algorithms. We demonstrate that, on a set of crafted routing instances, `DRouter` has substantially better capacity than either plain reduction to bit-vector reasoning or `Monosat`, a solver that is able to reason about SAT and graph predicates. We show that `DRouter` can route large clips from Intel designs while obeying up to millions of applications of the design rules–a task two industrial routers failed to accomplish.**

I. Introduction

Wire routing (or, simply, *routing)* is an essential stage in the process of the physical design of integrated circuits [17] and printed circuit boards [1]. A router *routes* (that is, connects) components laid out during the placement stage. *Design rules* specify restrictions on the routing process originating in manufacturing requirements.

Plain routing (that is, routing without design rules) is often modeled as the Steiner tree packing problem [1], [4], [8], [9], defined as follows: Let $G = (V, E)$ be a positively weighted simple graph. Let $N_{i \in \{0 \dots m-1\}} \subseteq V$ be m pairwise disjoint non-empty subsets of G's vertices, called the *nets*, where the vertices of each net are called the *terminals*. A *routing* is a forest of *net routings* $E_{i \in \{0 \dots m-1\}} \subseteq E$, such that $(V(E_i), E_i)$ is a tree that spans all N_i's terminals, where all the net routings are pairwise vertex-disjoint and the *optimization requirement* of minimizing the routing's total weight is met. An example is provided in Fig. 1. To solve plain routing, heuristic approaches, relaxing the optimization requirement to some extent, are commonly applied – see [1] for a survey.

In practice, routing must conform to design rules. For example, the *short rule* [18] states that no edge may touch two distinct net routings.

This paper extends the plain routing formulation to model design-rule applications. We let the user provide a SAT or bit-vector instance F along with a plain routing instance, where F's Boolean variables include the edges. The router must satisfy F, while guaranteeing that the satisfied edges comprise a routing. We refer to the resulting problem as *Routing Under Constraints (*RUC*)*.

It is well-known that plain routing (in various flavors) can be reduced to SAT [6], [19]. It has also been observed that design rules can be reduced to SAT [16], [18]. The apparent advantage of any SAT-based router to a heuristic router is that SAT can handle arbitrary constraints (corresponding to applications of arbitrary design rules) efficiently based on its sophisticated conflict analysis. In addition, modeling a new design rule for a heuristic router involves non-trivial modification of the router, whereas in a SAT-based approach any design rule reducible to SAT does not require modification of the router. Furthermore, unlike any heuristic router, a SAT-based router is complete.

The main challenge for any SAT-based approach to routing is scalability. As we shall see, a straightforward reduction of RUC to SAT through bit-vector reasoning does not scale.

To overcome the scalability issue, we designed a RUC solver, called `DRouter`. Essentially, `DRouter` implements a router inside a SAT solver. It overrides the SAT solver's decision and restart strategies with routing-aware strategies and enhances its learning with routing-aware conflict analysis.

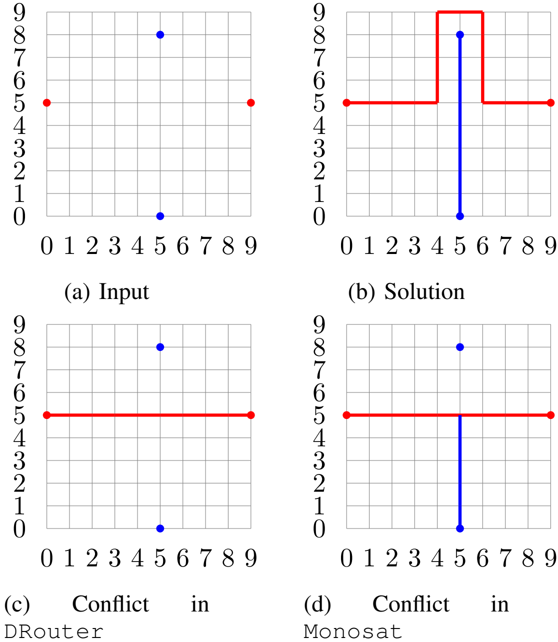
The usefulness of applying a graph-aware decision strategy and conflict analysis inside a SAT solver (in contrast to full reduction to bit-vector reasoning) was advocated and highlighted in a recent work on solving the NP-hard problem of finding a bounded-path (that is, a path whose weight falls within a user-given range) in a graph [7]. In [7], the decision strategy replaces the majority of the constraints; it guides the solver towards the solution, while taking additional optimization requirements into account. The decision strategy's role in our work is no less prominent.

`Monosat` [3] is a recent tool which can reason about graph reachability and bit-vectors. The RUC problem can be easily formulated in `Monosat` language. Let *Pathfinding under Constraints (*PFUC*)* be a restriction of RUC to one 2-terminal net. `DRouter`'s PFUC solving algorithm is conceptually similar to that of `Monosat`. We shall see that `DRouter` is substantially more efficient than `Monosat` for the generic RUC problem. Sect. V provides a detailed comparison between `Monosat` and `DRouter`.

We shall show that `DRouter` can route large clips of Intel design while obeying up to millions of applications of the design rules, whereas two industrial routers failed to do so.

In what follows, Sect. II contains preliminaries. Sect. III introduces our PFUC solving algorithm, called `DPF`. `DRouter`, introduced in Sect. IV, uses `DPF` as the underlying building

block. Sect.V compares `DRouter` to `Monosat`. Experimental results are presented in Sect. VI. Sect. VII concludes our work.



(a) Input (b) Solution

(c) Conflict in `DRouter` (d) Conflict in `Monosat`

Fig. 1: Plain routing example on a 10×10 solid grid graph, given two nets of two terminals each $N_0 = \{(0,5),(9,5)\}$ and $N_1 = \{(5,0),(5,8)\}$. Assume the edges' weights are 1.

II. Preliminaries

A. Bit-vector Reasoning and SAT

A bit-vector (BV) solver decides formulas comprising fixed-sized bit-vectors, Boolean variables and a variety of bit-vector and Boolean operators. An eager BV solver works by preprocessing the given BV formula, bit-blasting it to Conjunctive Normal Form (CNF) and solving with a SAT solver. We assume that the reader is familiar with the basics of SAT and eager BV solving. See [10] for a recent overview.

B. Modeling of Routing Under Constraints (RUC)

Our basic RUC modeling extends the plain routing modeling, presented in Sect. I. Let $G = (V, E)$ be a graph, $N_{i \in \{0 \ldots m-1\}} = \left\{ t_0^i, t_1^i \ldots t_{|N_i|-1}^i \right\} \subseteq V$ be the pairwise disjoint nets and $F(E \cup U)$ be a bit-vector formula (where U is a set of bit-vector and Boolean variables). Given a model for F, α, let $E^\alpha \subseteq E$ be a subset of edges assigned to 1 in α. The problem of *Routing Under Constraints (*RUC*)* is about finding a model α for F, such that any two terminals of the same net N_i are connected in $(V(E^\alpha), E^\alpha)$ and any two terminals of two different nets are disconnected in $(V(E^\alpha), E^\alpha)$.

Note that we leave out the requirement that the routing solution be a forest of *trees* and also the optimization requirement as to the overall weight. Similarly to heuristic routers, our algorithm will strive to heuristically reduce the overall weight of the satisfied edges.

For convenience, we extend our modeling as follows.

First, we let F use the vertices V (along with the edges) as Boolean variables. Given a RUC model α, an edge or vertex $b \in V \cup E$ is *active* iff $\alpha(b) = 1$.

Second, for every vertex v, we introduce a bit-vector variable $0 \leq nid(v) < m$ of width $\lceil log_2 m - 1 \rceil$ to represent the unique net id of active vertices, where the *net id* of net N_i is the index i. For each vertex v, we encode the *net boundary* constraint $0 \leq nid(v) < m$ into F.

Third, we encode the following *edge consistency* constraints into F: for each edge $e = (v, u)$, $e \implies v \wedge u \wedge (nid(v) = nid(u))$.

The short rule, mentioned in Sect. I, can now be encoded as follows: for each (not necessarily active) edge $e = (v, u)$, $v \wedge u \implies (nid(v) = nid(u)))$. Note that the short rule does *not* hold for the synthetic example in Fig. 1b.

C. Routing Complexity

Routing is a difficult problem. Plain routing is NP-hard even for one net [11] (in which case it is reduced to the classical problem of finding a Steiner tree). For multiple nets, plain routing is NP-complete even without the optimization requirement [13]. RUC is NExpTime-hard, since BV logic is trivially reducible to RUC, and BV logic is NExpTime-hard [12] (although various subsets of BV logic that are relevant in practice are NP-complete [5]).

D. Reducing RUC to Bit-Vector Reasoning

The following BV encoding for RUC can be applied if the nets are restricted to two terminals only. Assuming the edge consistency constraints are encoded (Sect. II-B), it remains to ensure that for each net i, its pair of terminals t_0^i and t_1^i is connected. That can be done by encoding the following cardinality constraint for each vertex: each terminal has one active neighbor edge and each non-terminal vertex has two active neighbor edges. One can extend the modeling to multi-terminal nets by encoding the construction of directed trees with a terminal sink for each net. We omit further details due to space restrictions. As we shall see, BV encoding does not scale well.

E. A* Algorithm

A* is a commonly used algorithm for finding a path in a graph from a source vertex s to a target vertex t, given an under-estimation, $h(v)$, of the weight from any vertex v to t. If $h(v) = 0$ for every v, A* operates identically to Dijkstra shortest-path algorithm. Having an accurate heuristical estimation helps A* converge faster.

We are interested in graphs having a grid-like structure, that is, graphs, whose vertices represent nodes of a two- or three-dimensional grid. This is because routing in the original physical design problem is carried out in a grid. One example of a grid-like graph is a *solid grid graph*, whose vertices correspond to the points of a two-dimensional grid and whose edges connect any two vertices at distance one (see the example in Fig. 1). Our algorithms apply Manhattan distance as the A* heuristic in a grid-like setting and set $h(v) = 0$ for each vertex v, otherwise.

We need a slightly modified version of A*.

First, the modified A* can receive an optional parameter: a set of *bad* edges and vertices *Bad*, which cannot be used for connecting s to t. The implementation of this feature is straightforward: whenever A* visits an edge or vertex $b \in Bad$, it abandons the exploration of the path containing b.

Second, the modified A* can receive another optional parameter: a set of edges, whose weight should be set to 0 for that particular A* invocation.

Third, in cases where s is not connected to t in G, given a set of bad edges and vertices A* returns a special value $\bot$ and generates a *conflict cut*. Intuitively, a conflict cut is a subset of the bad edges and vertices that prevented A* from connecting s to t. An empty conflict cut would mean that s is disconnected from t in G, independently of the bad set. Below, we provide a more precise definition of the conflict cut.

1) Let *Visited* $\subseteq V \cup E$ be the set of vertices and edges visited by A* during the graph traversal (that is, vertices and edges connected to s, *Bad* included)
2) Let *Unvisited* $\subseteq V \cup E$ be the set of all vertices and edges, unvisited by A*
3) Let the *frontier Frontier* $\subseteq V \cup E$ be a set, including:
   a) All the vertices from *Visited* which have at least one neighbor edge in *Unvisited*, and
   b) All of the edges from *Visited* whose other vertex is in *Unvisited*
4) The *conflict cut* is the set of vertices and edges that belong to both *Bad* and *Frontier*.

III. Pathfinding under Constraints

Let the *Pathfinding under Constraints (*PFUC*)* problem be RUC, restricted to one 2-terminal net. We propose an algorithm for solving PFUC, called DPF.

Given a graph $G = (V, E)$, a single net $N = \{s, t\}$ and a BV formula $F(E \cup V \cup U)$, DPF should return either a model α for F, such that there exists a path from s to t in $(V(E^\alpha), E^\alpha)$, or UNSAT, if no such model exists. DPF is designed to heuristically reduce the overall weight of the satisfied edges.

For the rest of this section, we assume that the constraint $e \implies v \wedge u$ for each edge $e = (v, u)$ is encoded into F. The net boundary and edge consistency constraints from Sect.II-B are unnecessary for pathfinding, since pathfinding involves a single net.

A. DPF *Algorithm*

DPF is implemented inside an eager BV solver's SAT solver. It overrides the SAT solver's decision strategy with an A*-based algorithm and records additional conflict clauses whenever s becomes disconnected from t because of propagation in F. DPF disables the SAT solver's restart strategy.

During DPF's invocation, an edge or vertex b can either be: *a*) *active*–if b is assigned 1, *b*) *inactive*–if b is assigned 0, or *c*) *unassigned*.

DPF's decision strategy tries to connect s to t by activating edges (that is, assigning 1 to edges) in a queue σ, where σ is initialized to the shortest path from s to t using A*. The activated edges of σ are moved from the front of σ to the back of the *actual path* stack π. Before each decision point, the concatenation $\pi \circ \sigma$ comprises a walk from s to t in G, where π's edges are active and σ's edges can be unassigned or active. When the algorithm is finished, π contains a walk from s to t. π might be a walk rather than a path for reasons which will be discussed later. We suggest a simple post-processing algorithm targeting cycle elimination in π in Sect. III-B.

```
1:  class PathFinder
2:      members:
3:          vertex s;                                   ▷ source vertex
4:          vertex t;                                   ▷ target vertex
5:          walk π = s ≡ π_0 π_1 ... π_{|π|-1} ≡ f;
6:          path σ = f ≡ σ_0 σ_1 ... σ_{|σ|-1} ≡ t;
7:      methods:
8:          Init(vertex s', vertex t', BV Formula F) → Is Conflict?
9:              s := s'; t := t'
10:             σ := A*(s,t)
11:             if A* returned ⊥ then return ⊥ else return ⊤

12:         Decide() → SAT literal
13:             if t ∈ π then                           ▷ π connects s to t
14:                 if Unassigned edge e exists then return ¬e
15:                 return SAT-Decide()
16:             return (σ_0, σ_1)

17:         Propagate() → Is Conflict?
Require: Invoked right after BCP if there was no conflict
18:             while |σ| = 1 or (σ_0, σ_1) is active do
19:                 Pop σ_0 from σ's front; push it to π's back
20:             if σ is not violated then return ⊤
21:             σ := A*(π_{|π|},t,inactive vertices and edges,active edges)
22:             if A* returned ⊥ then
23:                 Add a clause comprising the conflict cut
24:                 return ⊥
25:             while |σ| = 1 or (σ_0, σ_1) is active do
26:                 Pop σ_0 from σ's front; push it to π's back
27:             return ⊤

28:         Backtrack()
Require: Invoked after completing SAT solver's backtracking
29:             while π contains unassigned edges do
30:                 Pop the latest vertex from π's back
31:             σ := {}                                 ▷ Clear σ
```

Fig. 2: DPF Solver for Pathfinding under Constraints.

We say that σ is *violated* when one of its vertices or edges becomes inactive.

Consider the class implementing DPF in Fig. 2. The members of the class (lines 3– 6) comprise s, t, π and σ. Consider also the DPF invocation trace example in Fig. 3.

The method Init (line 8) is invoked before the SAT solver is launched. In Init, DPF initializes s and t and then checks whether there exists a path from s to t in G using A*. If no such path exists, the problem is unsatisfiable. Otherwise, the algorithm stores the path in σ. The modified SAT solver is then invoked. Fig. 3a illustrates the situation after initialization for

our example.

The method DECIDE (line 12) overrides the SAT solver's decision heuristic. If the target is already reached (that is, $t \in \pi$), DECIDE deactivates (that is, assigns 0) an unassigned edge, if any, to minimize the overall active edge weight. If all the edges are assigned, the algorithm invokes the default SAT decision strategy. If the target is not reached, DECIDE simply returns the first σ's edge (σ_0, σ_1). The methods PROPAGATE and BACKTRACK guarantee that (σ_0, σ_1) is unassigned when DECIDE is invoked.

Consider the method PROPAGATE (line 17). It is invoked right after the SAT solver's BCP (Boolean Constraint Propagation) if BCP did not encounter a conflict. Similarly to BCP, PROPAGATE should return whether it found a conflict ($\top$ and $\bot$, respectively, are returned for "no conflict" and "conflict").

PROPAGATE starts by moving any active edges from σ's front to π's back. Afterward PROPAGATE returns $\top$ if σ is not violated. We call the situation when σ is violated *path violation*. A path violation occurs in Fig. 3b of our example.

In case of path violation, we run A* so as to create a new σ to continue the path π towards t using active and unassigned edges only. We also set the weight of active edges to 0 for A*, in order not to "pay" for a single edge more than once in case A* reuses already assigned edges. This can happen, for example, if there is no path from $\pi_{|\pi|-1}$ to t anymore as is the case in Fig. 3b. This also explains why π might be a walk, rather than a path.

If A* was successful in finding a new σ, PROPAGATE moves any active edges from σ's front to π's back and returns. This is the case in our example. Fig.3c illustrates the situation just after PROPAGATE's completion.

Assume now that A* could not find a new path. This is the case in Fig. 3d continuing our example. In this case the algorithm adds the conflict cut as a conflict clause and returns $\bot$. The conflict cut is a subset of the inactive vertices and edges that blocks any path from s to t. In our example, the conflict cut comprises the following set $\{(2,0),(3,1)\}$. The conflict cut, seen as a clause, must always be falsified by the current partial assignment, since all its members are inactive. Hence recording it as a conflict clause triggers SAT solver's conflict analysis.

During conflict analysis, the solver will learn the 1UIP conflict clause and backtrack, so as to have one and only one asserting literal of the conflict clause unassigned. The method BACKTRACK (line 28) is invoked whenever SAT solver's backtracking is completed. The method aligns π with the current partial assignment (by popping all the unassigned edges from π) and clears σ. After backtracking, the solver flips the asserting literal and applies BCP, followed by a PROPAGATE invocation. Note that σ is populated again by PROPAGATE.

In our example, the learned 1UIP conflict clause is $\neg(2,0) \vee \neg(3,2)$ (we omit derivation details due to space constraints). The situation after backtracking, propagating and finding a new σ in our example is shown in Fig. 3e. The vertex $(3,2)$ is rendered inactive because of BCP in the new clause.

This completes the description of our PFUC algorithm. Fig. 3f shows a completed routing for our example.

As we mentioned, after completing the routing, the solver deactivates any unassigned edges to reduce the weight and then falls back to the default decision heuristic.

Note that even after the initial routing has been completed, the solver might still backtrack and change the routing. In our example in Fig. 3, replacing the clause $\neg(3,2) \vee \neg(3,1)$ by an equivalent set of clauses $\neg(3,2) \vee \neg(3,1) \vee x \vee y$, $\neg(3,2) \vee \neg(3,1) \vee \neg x \vee y$, $\neg(3,2) \vee \neg(3,1) \vee x \vee \neg y$, $\neg(3,2) \vee \neg(3,1) \vee \neg x \vee \neg y$, where x, y are auxiliary variables, would cause the solver to generate a "bad" routing through the vertices $(3,2)$ and $(3,1)$ which it would only fix later following conflict analysis and backtracking.

B. Optimization with the Decision Strategy

As we have seen, `DPF` applies the following two techniques as part of its decision heuristic to heuristically reduce the routing weight: *a*) using the shortest path A* algorithm, and *b*) deactivating any unassigned edges after the routing is completed.

In addition, one can apply the following post-processing algorithm to eliminate cycles in π (if any) to reduce the total edge weight. The algorithm below reuses the SAT solver instance created by `DPF`. The instance is updated and invoked incrementally. First, assuming `DPF` returned a model α, identify a simple tree in $(E^\alpha, V(E^\alpha))$ and provide its edges as unit clauses to the SAT solver. Second, run a plain SAT solver with the following single modification to the decision heuristic: deactive unassigned edges first.

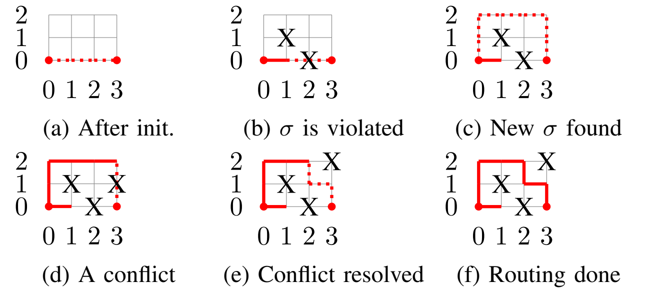


Fig. 3: `DPF` trace example. Assume that $s = (0,0)$ and $t = (3,0)$ and that the following CNF is provided: $\neg(1,0) \vee \neg(2,0)$, $\neg(1,0) \vee \neg(1,1)$, $\neg(3,2) \vee \neg(3,1)$. Dotted edges correspond to σ, while bold edges correspond to π. "X" marks inactive vertices.

IV. ROUTING UNDER CONSTRAINTS

This section introduces `DRouter` – our RUC solution. Similarly to `DPF`, `DRouter` is implemented inside a SAT solver. Sect. IV-A below adjusts `DPF` to routing. Our basic `DRouter` algorithm is presented in Sect. IV-B. Sect. IV-C, Sect. IV-D and Sect. IV-E introduce three enhancements to the basic algorithm which are crucial for scalability.

A. Routing-Aware Pathfinding

We need to make simple yet essential modifications to `DPF` to make it routing-aware. Our goal is to be able to apply `DPF` inside `DRouter` to connect terminals within a net.

First, net boundary and edge consistency constraints must be applied (recall Sect. II-B).

Second, the class PATHFINDER should have an additional member *nid*, representing the net id of the routed net, which will be initialized during INIT to a new (third) parameter.

Third, DECIDE will not be invoked after s is connected to t; hence the code in Fig. 3 between lines 13– 15 can be ignored.

Fourth, a crucial modification should be applied to PROPAGATE when a path violation is identified (line 21). When this happens, A* is invoked to find a new path to t. We disallow A* from using the vertices of any net other than the current net (in addition to the disallowed inactive vertices and edges). After this change, line 21 looks as follows:

σ := A*($\pi_{|\pi|}$,t,inactive vertices and edges $\cup$ any active vertex v, such that $nid(v) \neq nid$, active edges)

Furthermore, the conflict clause, created at line 23, will contain additional literals as follows: any vertex v of net id $nid(v) \neq nid$ will contribute to the conflict clause one bit on which the values of *nid(v)* and *nid* differ.

For example, assume that the modified DPF is invoked to connect the two terminals of N_1 after N_0 is routed as shown in Fig. 1c. A* will fail, since all the possible paths are blocked by N_0. The conflict clause will contain the only bit of *nid(v)* for every vertex v of row 5.

B. The Basic Algorithm

We can now present the basic algorithm of DRouter.

A class implementing the algorithm for 2-terminal nets is depicted in Fig. 4 (an extension for multi-terminal nets is proposed later in this section). The class maintains an array of net ids *nids*, the current index to *nids*, and an array of the class PATHFINDER. We assume that PATHFINDER is modified to be routing-aware as explained in Sect. IV-A.

The idea is simply to route the nets one by one using a fresh *pathfinder* (that is, a fresh instance of the class PATHFINDER) for each net.

The method INIT initializes *nids* and then initializes a pathfinder for net id 0.

The method DECIDE simply applies the DECIDE method of the currently routed net, if such exists. If all the nets are routed, DECIDE deactivates unassigned edges, if any, and, otherwise, lets the SAT solver take the decision.

PROPAGATE operates in a loop as long as non-routed nets exist. Within the loop, it tries to propagate in the currently routed net, which might result in a conflict, in which case PROPAGATE returns $\bot$. Otherwise, if the net is not yet routed, the method returns $\top$. If the net is routed, PROPAGATE initializes a pathfinder for the next net, if any, pushes it to the pathfinders vector and continues the main loop.

BACKTRACK backtracks over any fully routed nets, and then it backtracks within the currently routed net (if one exists).

Our algorithm can easily be extended to treat any multi-terminal net by connecting routing previously created for that net to any new terminal until the net is fully routed. A* can be applied, without modifications, for connecting multiple sources to a single target.

```
1:  class DRouter
2:    members:
3:      number[] nids;
4:      number currNidInd;
5:      PATHFINDER[] pfs;

6:    methods:
7:      INIT() → Is Conflict?
Require: SAT solving has not yet started
8:        nids := {0, 1, ..., m − 1}
9:        PATHFINDER p;
10:       if P.INIT(t_1^0,t_2^0,0) == ⊥ then return ⊥
11:       Push p to the back of pfs
12:       currNidInd := 0;
13:       return ⊤

14:     DECIDE() → SAT literal
15:       if currNidInd ≤ m then
16:         p := back of pfs
17:         return P.DECIDE()
18:       if Unassigned edge e exists then return ¬e
19:       return SAT-DECIDE()

20:     PROPAGATE() → Is Conflict?
Require: Invoked right after BCP if there was no conflict
21:       while currNidInd < m do
22:         p := back of pfs
23:         if P.PROPAGATE() == ⊥ then return ⊥
24:         if p.t ∉ p.π then return ⊤
25:         currNidInd := currNidInd + 1
26:         if currNidInd == m then return ⊤
27:         PATHFINDER p;
28:         n := nids[currNidInd]
29:         if P.INIT(t_1^n,t_2^n,n) == ⊥ then return ⊥
30:         Push p to the back of pfs

31:     BACKTRACK()
Require: Invoked after completing SAT solver's backtracking
32:       while currNidInd ≥ 0 do
33:         p := back of pfs
34:         P.BACKTRACK()
35:         if |p.π| > 1 then return
36:         Pop from pfs's back
37:         currNidInd := currNidInd − 1
```

Fig. 4: Basic DRouter for 2-terminal nets.

One can also apply a post-processing algorithm for cycle elimination to reduce the overall solution weight, similarly to Sect. III-B

C. Early Net Conflict Detection

Let *net conflict* be a situation during DRouter invocation when a certain *conflicting net* N_i cannot be routed anymore, since there exists no path in the graph between two of its terminals t_q^i and t_w^i ($q \neq w$).

Our algorithm might encounter a net conflict when A* is

applied to route a new terminal of the conflicting net. A net N_j *blocks* the conflicting net, if any vertex v, which belongs to N_j (that is, a vertex v, such that *nid(v)* $= j$), is part of the conflict cut.

A net conflict situation appears in Fig. 1c, where net N_0 blocks the conflicting net N_1.

Our basic algorithm identifies net conflicts only when it gets to routing the conflicting net, but it is desirable to discover and handle such situations earlier. By "handling" we mean recording a clause, which will cause the solver to backtrack and re-route.

We propose the following *early net conflict detection* algorithm. After any net is routed, we check, for every unrouted net, if it can still be routed by applying A* for connecting terminal i to terminal 0, for every $i > 0$. If a conflicting net is discovered, we record a conflict clause comprising the conflict cut found by A*.

To speed things up, one can keep, for each net and each terminal $i > 0$, a pre-routed path β to terminal 0 of that net, and check if β is still not violated before invoking A*. If A* has to be invoked and it finds a path, β can be updated to that path.

D. Net Swapping

Net ordering is crucial for our algorithm. Consider the example in Fig. 1a. If `DRouter` picks N_1 as the first net, the solution in Fig. 1b will instantly be found without any net conflicts. Picking N_0 as the first net would result in a net conflict, shown in Fig. 1c. The algorithm described so far would have to record conflict clauses to disqualify a variety of paths connecting N_0's terminals until it discovers a path which does not block N_1. Such an approach might cause run-time and/or memory explosion issues in practice.

We propose two solutions for this problem. The first is net swapping, presented next. The second is net restarting, presented in Sect. IV-E.

Net swapping is applied after a net conflict is discovered and the corresponding conflict clause recorded. Assume N_i is the contradicting net. It might be blocked by several previously routed nets. Let N_j be the net routed last out of all nets blocking N_i. In that case, the *nids* array, which defines routing order, should appear as follows: $\{A, j, B, i, C\}$ (where A, B, and C each represent a sequence of net ids). Net swapping backtracks to the decision level just before the algorithm started routing N_j and swaps between the nets N_j and N_i. In addition, it moves N_i to immediately follow N_j. *nids* will look as follows after net swapping: $\{A, i, j, B, C\}$. Hence, after net swapping, the algorithm will attempt to route N_i, followed by N_j.

Net swapping solves the problem in Fig. 1a even if N_0 is picked as the first net simply by swapping the nets after the first net conflict.

Net swapping might not be sufficient for finding a routable net ordering, because it is restricted to two nets only. For example, a problem might occur if the two nets block each other, regardless of the order, because of previously routed nets. In such a case, the algorithm will keep swapping the two nets. The algorithm would still complete because of conflict clause recording, but it might be inefficient. Net restarting, presented next, is another, more global, algorithm for changing the net ordering, based on information derived during net conflict analysis.

E. Net Restarting

Net restarting is the following simple yet effective technique for net conflict-aware net reordering.

We associate a *conflict counter* with each net which is increased whenever the net becomes conflicting in a net conflict. Once the counter reaches a user-given threshold T (10 by default) for some N_i, `DRouter` restarts and places i before all the other nets in *nids*, so as to start routing N_i right after the restart. The conflict counters for all nets are set to 0 following a net restart.

Assume that N_0 is the first net in our example in Fig. 1a. Applying net restarting alone (without net swapping) will solve the problem after T net conflicts by restarting, placing N_1 before N_0, and routing.

Sect. VI will demonstrate that combining net swapping and net restarting yields the best results in practice.

V. COMPARING `DRouter` TO `Monosat`

`Monosat` [3] is a recent solver that can reason about a BV formula F and various graph predicates, given one or more graphs sharing edges with F. In particular, `Monosat` can reason about graph reachability predicates, where a reachability predicate *reach(v,u)* holds iff vertex v is connected to vertex u through active edges (that is, edges assigned 1).

The RUC problem can easily be modeled in `Monosat` as follows. First, the BV formula F and the graph G are provided as input to `Monosat`. Second, a reachability predicate *reach*(t_0^k,t_i^k) is created and globally asserted for each terminal t_i^k for $i > 0$ for each net N_k. That guarantees that each net N_k is routed. Third, a reachability predicate *reach*(t_0^k,t_0^l) is created and its *negation* $\neg$*reach*(t_0^k,t_0^l) is globally asserted for each pair of the first terminals t_0^k and t_0^l for each pair of nets N_k and N_l for $k < l$. That guarantees that all the nets are routed disjointly.

`Monosat` takes advantage of dedicated conflict analysis techniques for reasoning about reachability predicates. It applies the Ramalingam-Reps incremental shortest path algorithm [15] to keep track of the status of reachability predicates. Whenever an asserted predicate *reach(v,u)* is violated, that is, whenever v and u are no longer connected through active edges, Monosat creates a conflict clause comprising the conflict cut, similarly to our `DPF` algorithm. Whenever an asserted negated predicate $\neg$(*reach(v,u)*) is violated, that is, whenever v and u become connected through active edges, `Monosat` records a conflict clause comprising the shortest path connecting v to u through active edges.

`Monosat` can also be configured to apply a dedicated

decision heuristic for globally asserted reachablity predicates.[1] Monosat's decision heuristic connects the vertex v to u, for each asserted predicate *reach(v,u)* in the user-given order, by shortest path, using the Ramalingam-Reps algorithm, similarly to our DPF algorithm, the difference being that DPF uses the cheaper A* algorithm lazily, whereas Monosat uses the incremental Ramalingam-Reps eagerly.

Let us compare the functionality of DRouter without net swapping and net restarting to that of Monosat with the decision heuristic on the very simple routing instance in Fig. 1a.

Assume that N_0 is routed first. Both Monosat and DRouter will easily route N_0. The key difference is that DRouter will identify a net conflict immediately after N_0 is routed, since N_0 blocks N_1 (see Fig. 1c), while Monosat will start routing N_1 and discover a conflict only when the nets become connected, as shown in Fig. 1d. Then Monosat will learn a clause consisting of all the active edges in the bottom-left sub-grid $(0, 0) - (5, 5)$ in Fig. 1d. Monosat will have to create an exponential number of clauses to falsify any single N_0 routing blocking N_1 (including that shown in Fig. 1c), whereas DRouter falsifies any single N_0 routing at once.

Assume now that N_1 is routed first. DRouter solves such a problem instantly without any conflicts. Monosat might still encounter an exponential number of conflicts before finding a solution, since after routing N_1 it will keep trying to route N_0 using its shortest path heuristic right through N_1 routing.

For these reasons, DRouter, even without the advanced techniques of net swapping and net restarting, is expected to be considerably more efficient than Monosat for the RUC problem. Net swapping and net restarting make DRouter substantially more efficient.

All in all, unlike DRouter, Monosat is not *routing*-aware, although it is *reachability*-aware. In addition, Monosat is less optimization-aware than DRouter as Monosat neither tries to deactivate unassigned edges nor has a post-processing optimization loop (recall Sect. III-B).

VI. Experimental Results

In this section, we describe various experiments on crafted and industrial instances. For all the experiments, the run-time is measured in seconds, the memory is measured in Gb, and "TO" stands for time-out.

A. Crafted Instances

This section presents experiments with crafted instances. Detailed results and all the benchmarks are publicly available at [14]. We used Intel® Xeon® CPU E3-1270 v3 machines with 32Gb of memory and 3.50GHz frequency. We set the time-out to 20 min. The following RUC solvers were used: *a*) DR: shortcut for DRouter, *b*) DR−S: DRouter without net swapping, *c*) DR−R: DRouter without net restarting, *d*) DR−SR: DRouter with neither net swapping nor net restarting, *e*) Mn: shortcut for default Monosat (version 1.2.0), *f*) Mn+D: Monosat with the decision heuristic for reachability (*-decide-theories* switch is applied), *g*) BV: Sect. II-D's reduction of RUC to BV and application of Intel's eager BV solver Hazel.

[1]The decision heuristic is not mentioned at all in the conference paper [3] and is only briefly mentioned in the paper's extended version [2]. We are grateful to the first author of [3] for sharing the details in private communication.

Size	First Net	DR	DR−S	DR−SR	DR−R	BV	Mn	Mn+D
10	N_0	0	0	0	0	0	55	TO
1000	N_0	25	31	TO	26	TO	TO	TO
10	N_1	0	0	0	0	0	222	TO
1000	N_1	25	25	25	25	TO	TO	TO

TABLE I: Run-time comparison on several crafted instances.

1) Basic Comparison: The goal of our first experiment is to confirm the conclusion of Sect. V that DRouter should scale much better than Monosat for RUC even on very simple instances without any constraints. To that end, we created two benchmarks comprising the RUC instance in Fig. 1 for the two different possible initial net orderings. We also created two instances for two net orderings for a larger benchmark, structurally similar to that in Fig. 1, comprising a 1000×1000 grid with two nets: $N_0 = \{(0, 500), (999, 500)\}$ and $N_1 = \{(500, 0), (500, 998)\}$. The results appear in Table I ("First Net" stands for the first net in the net ordering).

Monosat with the decision heuristic (Mn+D) cannot solve a single instance, whereas DRouter with either net swapping or net restarting (or both) enabled instantly solves both 10×10 instances and easily solves both 1000×1000 instances. This result confirms our analysis in Sect. V.

Interestingly, DRouter without net swapping and net restarting (DR−SR) can easily solve the 10×10 instances, but can solve the 1000×1000 benchmark only when N_0 is routed first, that is, when there are no net conflicts. This is because when N_1 is routed first, there are too many conflict clauses to record for DR−SR.

Default Monosat can solve the 10×10 benchmark, but it is substantially slower than the other solvers. Hence, it comes as no surprise that default Monosat cannot solve the 1000×1000 benchmark.

2) Extended Comparison: We now present experimental results on RUC benchmarks crafted as follows (where $N = 20$):

```
1: for all M ∈ {3, 5, 7} do
2:     for all C ∈ {0, 10, 20, 30} do
3:         Generate a solid grid graph having N ∗ M rows
           and columns
4:         Create N 2-terminal nets, with randomly picked
           terminals
5:         Let V = (N ∗ M)^2 be the number of vertices.
           Generate C/100 ∗ V binary clauses as follows. Pick a
           random vertex v = (x, y) and another random vertex u
           sharing either x or y coordinate with V. Add the clause
           ¬v ∨ ¬u.
```

Note that M regulates the grid size and C regulates the number of generated clauses. We created 10 instances for each $M \times C$ combination.

Consider Table II. DRouter is the only solver able to solve all the instances. Monosat in either mode, BV, and

M	C	DR	DR-S	DR-SR	DR-R	BV	Mn	Mn+D
3	0	10	10	0	0	0	0	0
3	10	10	0	0	10	0	0	0
3	20	10	0	0	0	0	0	0
3	30	10	0	0	0	0	0	0
5	0	10	10	0	0	0	0	0
5	10	10	10	0	0	0	0	0
5	20	10	10	0	0	0	0	0
5	30	10	10	0	10	0	0	0
7	0	10	10	0	0	0	0	0
7	10	10	0	0	0	0	0	0
7	20	10	0	0	0	0	0	0
7	30	10	0	0	0	0	0	0

TABLE II: Comparison of the number of solved crafted instances.

Area in μm^2	Nets	Vertices	Constraints	Time	Memory
24	110	42,456	484,008	25	0.7
24	230	42,456	484,008	391	1
32	352	63,740	667,764	705	2.2
129	788	127,480	2,669,056	14,733	6.5
129	891	127,480	2,669,056	92,950	6.5

TABLE III: `DRouter` performance on industrial instances. "Constraints" represent the number of design rule applications.

`DR-SR` cannot solve a single instance. Some instances are solved solely with net swapping and some others are solved solely with net restarting, but only their combination renders `DRouter` scalable.

B. Industrial Instances

This section shows that `DRouter` can scale to large clips from Intel designs which could not be routed by two modern industrial routers without violating design rules.

Consider Table III. We used Intel® Xeon® CPU E7-4870 machines with 2.40GHz frequency and 528Gb of memory. `DRouter` solves large industrial instances having hundreds of nets and up to millions of design rule applications, where the number of vertices reaches into the hundreds of thousands.

Two industrial routers we tested failed to route these clips. First, a typical heuristic router was only able to find routings that violated some of the rules. Second, an incomplete router based on enumerating some of the potential solutions and then picking an actual solution out of the potential ones using a SAT solver [16], failed to route these clips due to memory-outs (despite the machines' having as much as 528Gb of memory).

These results demonstrate that `DRouter` gives clear added value in industrial settings.

VII. Conclusion

This paper proposed a formal model for the problem of design-rule-aware routing. Our model combines graph theory (for representing the routing problem) and bit-vector logic (for representing applications of the design rules). We introduced a solver for the resulting problem, called `DRouter`. Essentially, `DRouter` implements an A*-based router inside a SAT solver, overriding the solver's decision and restart strategies and enhancing its learning with routing-aware algorithms. We demonstrated that `DRouter` has substantially better capacity than either plain reduction to bit-vector reasoning or the `Monosat` solver. Furthermore, we showed that `DRouter` can route large clips from Intel designs while obeying up to millions of design rule applications–a task two industrial routers failed to accomplish.

VIII. Acknowledgments

We are grateful to Kostas Malinauskas for carrying out the experiments, presented in Sect. VI-B. We thank Suto Gyuszi, Nina Lane and Kostas Malinauskas for many useful discussions. We are grateful to Sam Bayless for his essential help with `Monosat` and to Paul Inbar for editing the paper.

References

[1] N. Abboud, M. Grötschel, and T. Koch. Mathematical methods for physical layout of printed circuit boards: an overview. *OR Spectrum*, 30(3):453–468, 2008.

[2] S. Bayless, N. Bayless, H. H. Hoos, and A. J. Hu. SAT modulo monotonic theories. *CoRR*, abs/1406.0043, 2014.

[3] S. Bayless, N. Bayless, H. H. Hoos, and A. J. Hu. SAT modulo monotonic theories. In B. Bonet and S. Koenig, editors, *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA.*, pages 3702–3709. AAAI Press, 2015.

[4] S. Chopra. Comparison of formulations and a heuristic for packing steiner trees in a graph. *Annals of Operations Research*, 50(1):143–171, 1994.

[5] N. Dershowitz and A. Nadel. Is bit-vector reasoning as hard as nexptime in practice? In *13th International Workshop on Satisfiability Modulo Theories*, 2015.

[6] S. Devadas. Optimal layout via boolean satisfiability. In *1989 IEEE International Conference on Computer-Aided Design, ICCAD 1989, Santa Clara, CA, USA, November 5-9, 1989. Digest of Technical Papers*, pages 294–297. IEEE, 1989.

[7] A. Erez and A. Nadel. Finding bounded path in graph using SMT for automatic clock routing. In D. Kroening and C. S. Pasareanu, editors, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*, volume 9207 of *Lecture Notes in Computer Science*, pages 20–36. Springer, 2015.

[8] P. J. Esrom. Combinatorial algorithms for integrated circuit layout. *Robotica*, 9(1):118, 1991.

[9] M. Grötschel, A. Martin, and R. Weismantel. The steiner tree packing problem in VLSI design. *Math. Program.*, 77:265–281, 1997.

[10] L. Hadarean. *An Efficient and Trustworthy Theory Solver for Bit-vectors in Satisfiability Modulo Theories*. Dissertation, New York University, 2015.

[11] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York.*, The IBM Research Symposia Series, pages 85–103. Plenum Press, New York, 1972.

[12] G. Kovásznai, A. Fröhlich, and A. Biere. On the complexity of fixed-size bit-vector logics with binary encoded bit-width. In P. Fontaine and A. Goel, editors, *10th International Workshop on Satisfiability Modulo Theories, SMT 2012, Manchester, UK, June 30 - July 1, 2012*, volume 20 of *EPiC Series*, pages 44–56. EasyChair, 2012.

[13] M. Kramer and J. van Leeuwen. The complexity of wire-routing and finding minimum area layouts for arbitrary VLSI circuits. *Advances in computing research*, 2:129–146, 1984.

[14] A. Nadel. Routing under constraints: Benchmarks and detailed results. https://goo.gl/OUXido.

[15] G. Ramalingam and T. W. Reps. An incremental algorithm for a generalization of the shortest-path problem. *J. Algorithms*, 21(2):267–305, 1996.

[16] N. Ryzhenko and S. Burns. Standard cell routing via boolean satisfiability. In P. Groeneveld, D. Sciuto, and S. Hassoun, editors, *The 49th Annual Design Automation Conference 2012, DAC '12, San Francisco, CA, USA, June 3-7, 2012*, pages 603–612. ACM, 2012.

[17] N. A. Sherwani. *Algorithms for VLSI physical design automation*. Kluwer, 3 edition, November 1998.

[18] B. Taylor and L. T. Pileggi. Exact combinatorial optimization methods for physical design of regular logic bricks. In *Proceedings of the 44th Design Automation Conference, DAC 2007, San Diego, CA, USA, June 4-8, 2007*, pages 344–349. IEEE, 2007.

[19] R. G. Wood and R. A. Rutenbar. FPGA routing and routability estimation via boolean satisfiability. *IEEE Trans. VLSI Syst.*, 6(2):222–231, 1998.