Finding Bounded Path in Graph Using SMT for Automatic Clock Routing

Amit Erez(✉) and Alexander Nadel

Intel Corporation, P.O. Box 1659, 31015 Haifa, Israel
{amit.erez,alexander.nadel}@intel.com

Abstract. Automating the routing process is essential for the semiconductor industry to reduce time-to-market and increase productivity. This study sprang from the need to automate the following critical task in clock routing: given a set of nets, each net consisting of a driver and a receiver, connect each driver to its receiver, where the delay should be almost the same across the nets. We demonstrate that this problem can be reduced to bounded-path, that is, the NP-hard problem of finding a simple path, whose cost is bounded by a given range, connecting two given vertices in an undirected positively weighted graph. Furthermore, we show that bounded-path can be reduced to bit-vector reasoning and solved with a SAT-based bit-vector SMT solver. In order to render our solution scalable, we override the SAT solver's decision strategy with a novel graph-aware strategy and augment conflict analysis with a graph-aware procedure. Our solution scales to graphs having millions of edges and vertices. It has been deployed at Intel for clock routing automation.

1 Introduction

Integrated circuits (IC) are made up of a large number of transistors forming logical gates connected by *nets*. The process of finding the geometrical layout of all the nets is called *routing*. Routing is an essential stage of the physical design process [25]. A clock is a control signal that synchronizes data transfer in the circuit. Specialized algorithms are required for routing the clock nets as opposed to other types of nets [26]. This is because the clock must arrive at all functional units at almost the same time. Clock nets must be routed before the other nets (except the power nets), hence rapid clock routing is critical for decreasing the time-to-market of semiconductor products. In clock routing, the following requirement must often be met for a set of nets, each net consisting of a driver and a receiver: wires connecting the driver to the receiver must have almost the same delay across the nets. This type of routing is called *matching constrained routing (MCR)*. This paper shows how to automate MCR.

Section 2 reviews related work and provides some preliminaries. We define the *bounded path problem* (or, simply, *bounded-path*) as follows: given a positively weighted undirected graph, a source s and a target t, find a *bounded path* (that is, a simple path, whose cost lays within a given cost range) from s to t. Section 3 shows that MCR can be reduced to bounded-path in a grid graph.

D. Kroening and C.S. Păsăreanu (Eds.): CAV 2015, Part II, LNCS 9207, pp. 20–36, 2015.
DOI: 10.1007/978-3-319-21668-3_2

Section 4 demonstrates that bounded-path is NP-hard even for a grid graph. It also shows how to reduce bounded-path to bit-vector (BV) logic. Solving bounded-path instances originating in MCR with a BV solver does not scale to industrial instances. Section 5 remedies this situation by proposing a new problem-aware approach to solving bounded-path within an eager BV solver [9,14]. First, we override the decision strategy of the SAT solver with a graph-aware strategy, which builds a bounded path from source to target explicitly. Second, we augment conflict analysis with graph-aware reasoning.

The main conceptual novelty of our approach w.r.t the decision procedure, independent of the particular problem, is the pivotal role of the decision strategy. While custom SAT decision heuristics have been applied previously [3,24], our decision strategy *replaces constraints*, that is, it guarantees that the algorithm is sound even after we remove the heaviest part of the constraints used in our initial reduction to BV logic. In addition, we use the decision strategy rather than constraints for heuristically optimizing the solution (w.r.t track utilization).

Furthermore, the underlying ideas behind graph-aware reasoning can be used to speed-up SAT-based approaches to other graph reachability problems, such as routing in the presence of design patterns [23] and cooperative path finding [28].

Section 6 of this work presents experimental results. We study the impact various aspects of our approach have on crafted bounded-path instances (available in [11]). In addition, we demonstrate that our approach solves a family of instances originating in the clock routing of modern Intel designs. Section 7 concludes our work.

2 Related Work and Preliminaries

The term clock routing is often associated with a routing scenario where the driver needs to be connected to *multiple* receivers within *the same net*, forming a tree wherein the delay from the driver to each receiver should be almost identical [13,31]. This scenario does not fall within the scope of this work.

The current solutions for MCR in IC [16,22] are designed for handling analog and mixed designs with exactly zero allowed skew (where *skew* is deviation in delay). The solution space explored in [16,22] is limited to cases where the number of wire segments in all nets is identical, and the length, layer and width of respective wires are identical. These limitations guarantee that the zero allowed skew requirement is met but are too restrictive for our setting. In particular, if the routing area is not rectangular (a common phenomenon in hierarchical designs with non-rectangular hierarchical block boundaries), none of the valid routing solutions are expected to conform to these limitations in a variety of test-cases. In addition, in our setting the allowed skew is greater than zero.

A propositional formula in Conjunctive Normal Form (CNF) is a conjunction/set of Boolean clauses, where each clause is a disjunction of literals and a literal is a Boolean variable or its negation. A SAT solver [8,20,27] receives a CNF formula and returns a satisfying assignment to its variables, if one exists. An eager BV solver [9,14] works by preprocessing the given BV formula [9,14,21],

bit-blasting it to CNF and solving with SAT. We assume that the reader is familiar with the basics of modern SAT and eager BV solving. See [17] for a recent overview.

Propositional satisfiability has been applied to solving the NP-complete problem of FPGA routing since [30]. From [30] we borrow the idea of using connectivity constraints to ensure that two given nodes are connected. The core problem in MCR of routing with almost the same delay does not exist in FPGA routing.

A DPLL(T) [15] theory solver for reasoning about costs to ensure that any satisfying assignment lays within some user-given cost bound has been proposed in [10]. Conceptually, the added value of our approach lies in: (a) introducing the concept of a decision strategy which replaces constraints and *guides* the solver towards a good solution while meeting additional optimization goals, and (b) introducing graph-aware reasoning.

We need some graph theory-related notations. Given an undirected graph $G = (V, E)$, where each edge $e \in E$ is associated with a positive cost c_e, a source node $s \in V$, a target node $t \in V$, and a simple path π from s to t of cost c (where the cost of a path is the sum of the costs of its edges), π is the *longest* path if there is no path from s to t of cost greater than c. π is *bounded* in the given cost range $[c_{min}, c_{max}]$, if $c_{min} \leq c \leq c_{max}$. A vertex $v \in V$ is *internal* if it is neither a source nor a target. We denote by $S = \sum_{e \in E} c_e$ the sum of the costs of all the edges in the graph. Let $m = (c_{max} + c_{min})/2$ be the middle of the cost range. Then the *actual skew* $k = |c - m|/(c_{max} - m)$ is the deviation of the generated path's cost from the middle. Sometimes the cost range is provided as a pair consisting of the *target cost* tc and the *allowed skew* a, which is equivalent to the cost range $[(1 - a) * tc, (1 + a) * tc]$.

We define a grid graph next. Let I be the infinite graph whose vertex set consists of all points of the plane with integer coordinates and in which two vertices are connected if the Euclidean distance between them is equal to 1. A *grid graph* is a finite, node-induced sub-graph of I. A vertex v in a grid graph is uniquely determined by its coordinates (v_x, v_y). A *vertical tracki* or a *horizontal tracki* comprises vertices whose x-coordinate or y-coordinate, respectively, is i. The maximal degree of a vertex in a grid graph is 4. An edge in a grid graph is either *vertical*, if the x-coordinates of its vertices are identical, or, otherwise, *horizontal*. The grid graph G is *mainly vertical* if most of its edges are vertical, otherwise it is *mainly horizontal*. Figure 4a on page 13 is an example of a mainly vertical grid graph. A vertex $v = (v_x, v_y)$ is to the `north`/`south`/`east`/`west` of $u = (u_x, u_y)$ if $v_y > u_y / v_y < u_y / v_x > u_x / v_x < u_x$, respectively.

Finally, we provide some relevant complexity results. Let *longest-path* be the problem of finding a longest path. Longest-path is NP-hard even for an unweighted graph, since Hamiltonian-path is trivially reducible to longest-path (see, e.g., [18]). Moreover, Hamiltonian-path, and thus longest-path, is NP-hard even for an unweighted grid graph [19]. Clearly, finding a longest path in a weighted graph and a weighted grid graph is also NP-hard. Longest-path is polynomial for some special grid graph classes, including *solid* grid graphs, where all of the bounded faces have area one (that is, the grid has no "holes") [29].

3 Matching Constrained Routing in Clock Routing

Let *net* be a subset of vertices in a 3-dimensional grid. *Routing* is about connecting all the vertices for each net with wires, where intersecting and/or touching wires which belong to different nets is not allowed (once the vertices have been connected the wiring is also considered to be part of the net).

Consider our more specific setting. Let $\{n_0, n_1, \ldots, n_k\}$ be a set of nets, where each net comprises the *driver* (source vertex) and the *receiver* (target vertex). First, as in any routing, in MCR one must connect the driver to the receiver for each net without intersection. Second, in MCR the delay must be similar for each net up to an allowed skew, where the *delay* is the amount of time it takes for the signal to travel from the driver to the receiver.

In our setting, the routing can use two adjacent x-y planes of the 3-dimensional grid only, where one plane is called the *horizontal metal* and the other is the *vertical metal*. The wires in the horizontal/vertical metal must lay along the horizontal/vertical tracks only, respectively. The two metals can be connected (with so-called *vias*). Superimposing the two metals reduces the problem space to a two-dimensional grid graph, where each intersection between available sub-tracks induces a vertex as shown in Fig. 1.

The routing delay depends on the length of the wires and the physical properties of the metals used. To model the similar delay requirement, we associate each edge with a cost proportional to the length of the wire represented by the edge, multiplied by a constant C_h or C_v, depending on whether the edge is horizontal or vertical. The ratio between constants C_h and C_v represents the difference in delay between the horizontal and vertical metals.

To generate routing with similar delay for the given set of nets, we proceed as follows. For each net independently we find the shortest path connecting its driver to its receiver. We then select as a *reference cost (RC)* the cost of the *longest* shortest path π_{rc} connecting the driver to the receiver for some net n_{rc}. π_{rc} comprises the solution for n_{rc}. Then, for each remaining net we formulate and solve a *separate* bounded-path instance, with the target cost being the RC and the allowed skew being user given (e.g., 2.5 %), where sub-tracks occupied by previously laid out nets are not part of the problem, as shown in Fig. 1. Hence the resulting grid graphs are normally not solid.

Moreover, as is the case with other routing algorithms, the router is requested to use as few tracks as possible. It is also desirable to minimize the actual skew. Both of these requirements are naturally translated into similar requirements for the bounded-path solver. Both are not strict in the sense that a good enough rather than the optimal solution is required.

4 Reducing Bounded-Path to Bit-Vector Reasoning

This section shows that bounded-path is NP-hard, and provides an encoding of bounded-path into BV logic. We start with Proposition 1, which shows that bounded-path is NP-hard by reducing longest-path to a binary search over

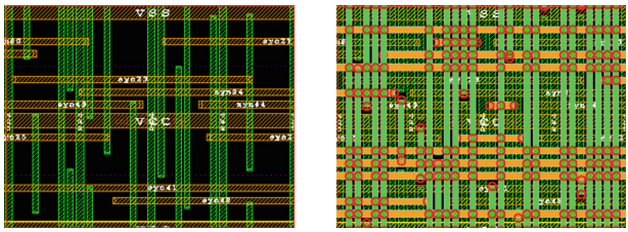


Fig. 1. Reducing the physical design problem (left) to a grid graph (right). On the left we see a bird's-eye view on a piece of layout with some of the vertical and horizontal tracks already occupied by wires. On the right we see the grid graph generation process. Legal sub-tracks are formed in non-occupied track parts, not too close to wires ends. Intersections and edge points of the legal sub-tracks are the vertices in the resulting grid graph. Edges are induced by the connections between the vertices.

the entire cost range, where each invocation solves bounded-path. Proposition 1 holds for any graph class for which longest-path is NP-hard, including weighted grid graphs induced by MCR. The extended version of this work [12] details the proof of Proposition 1 and provides lower-level examples of our encoding, which is introduced next.

Proposition 1. *The bounded path problem is NP-hard.*

We propose a reduction of bounded-path to bit-vector (BV) logic. Given an instance of bounded-path, our encoding ensures that a BV solver will output SAT and return a bounded path iff such exists.

We call an edge/vertex *active* iff it appears on the path from s to t and *inactive* otherwise. Consider now Fig. 2.

First, we associate each edge e and vertex v with a Boolean variable a_e and a_v, respectively, to represent whether the edge or the vertex, respectively, is active (items 1a and 1b in Fig. 2). The set of active vertices and edges comprise the bounded path returned by the solver for a satisfiable problem. The variables c_v and dir_e, discussed next, are intended to contain meaningful values for active edges and vertices only.

Second, each vertex v is associated with a BV variable c_v which represents the cost of the path from the source s to v if v is active (item 2a in Fig. 2). The width of c_v for each v is set to $\lceil log_2 S \rceil + 1$ to be able, in the worst case, to accommodate the cost of all the edges S without overflow.

Third, each edge e is associated with a Boolean variable representing its direction dir_e (item 2b in Fig. 2). The direction dir_e is intended to contain 0 for $e = (v, u)$ iff v is closer to s than u on the constructed path from s to t, in which case we say that e is *v-outgoing* and *u-incoming*. The other option is that $dir_e = 1$, and we say that e is *u-outgoing* and *v-incoming*.

Next, we introduce the constraints. They can be classified into connectivity constraints and cost constraints.

Connectivity constraints guarantee that a valid path of an arbitrary cost from s to t is constructed by the solver. Given a vertex v, let the set of v's *neighbors* be the set of edges touching v. Constraint 3a ensures that if an edge $e = (v, u)$ is active, then both v and u are active, while constraint 3b ensures that each vertex has a proper number of active neighbors. Specifically, an inactive vertex has no active neighbors. The source and the target vertices have one active neighbor each, while an internal active vertex has two active neighbors.

Figure 2 contains a high-level representation of constraint 3b's encoding. The actual encoding requires a solver supporting *conditional cardinality constraints* of the form $a \rightarrow \texttt{exactly}_k N$ (that is, if a Boolean a holds, then exactly k out of the set of Boolean variables N hold), where k is either 0, 1, or, 2 and N can be as large as the maximal vertex degree. While such constraints are not part of the standard BV language [4], an eager SMT solver can easily be extended to support them. This can be done by encoding the cardinality constraint $\texttt{exactly}_k N$ as a set of clauses (the problem is well-studied; see [7] for an overview) and then adding the selector literal $\neg a$ to each clause. Note that in a grid graph, the maximal degree of any vertex is 4, hence conditional cardinality constraints can be expressed with just a few clauses.

Consider now the cost constraints. They ensure that the cost of the constructed path falls within the specified cost range.

Constraints 4a to 4c guarantee that the direction is set correctly for any active edge. Namely, constraint 4a ensures that the active edge touching the source s must be s-outgoing, while constraint 4b ensures that the active edge touching the target t must be t-incoming (note that connectivity constraints guarantee that there is one and only one active edge touching the source and the target). Constraint 4c guarantees that if an internal vertex v is active, it has one v-incoming and one v-outgoing edge.

Finally, constraints 4d to 4f ensure that the eventual cost falls within the specified range. The cost is 0 for the source (constraint 4d) and it falls within the user-given range for the target (constraint 4f). The cost is propagated through the path's vertices taking advantage of the fact that the previous vertex is available through the direction of the incoming edge (constraint 4e).

5 Graph-Aware Solving

This section introduces graph-aware reasoning that enhances the eager approach to BV solving. In our new approach, the BV solver is provided with the connectivity variables and constraints only; the cost variables and constraints are omitted, thus substantially reducing the size of the problem. Our graph-aware decision strategy ensures that the path returned will still be bounded.

5.1 Graph-Aware Solving with Augmented Conflict Analysis

Consider Algorithm 1 which comprises the algorithmic framework of our approach. The algorithm contains five functions:

Algorithm 1. Graph-Aware Solving

```
1: function Solve(Graph G, Source s, Target t, Cost c_min, Cost c_max)
2:     For every vertex v, compute the minimal cost m(v) to t with Dijkstra
3:     Generate connectivity constraints and bit-blast to SAT
4:     tc:=(c_max + c_min)/2
5:     P:= []; l:=s; curr_cost:=0; stage:=init
6:     loop
7:         s:= Run the SAT solver
8:         if s = SAT then
9:             return P
10:        else if s = UNSAT then
11:            return No path exists
12:        else                                              ▷ s = UNKNOWN
13:            Refine by providing the clause ¬P to the SAT solver and restart the SAT solver

14: function OnDecision(Decision level d)
15:    if stage ≠ shortestp and curr_cost + m(l) ≥ tc then
16:        stage:=shortestp
17:    if stage = shortestp then
18:        N:= unassigned edges in nbors(l)
19:        return e ∈ N minimizing c_e + m(other_ver(e, l))
20:    e:= NextEdge
21:    l:=PathPushBack(e,d)
22:    return a_e

23: function OnImplication(Literal l)
24:    if l ≡ a_e for e = (l, v) then
25:        l:= PathPushBack(e, not_a_decision)

26: function OnBacktrack(Decision level d)
27:    {P, l, curr_cost, stage} :=backtrack_point(d)

28: function PathPushBack(Edge e, Decision level d)
29:    if d ≠ not_a_decision then
30:        backtrack_point(d):= {P, l, curr_cost, stage}
31:    curr_cost:=curr_cost + c_e
32:    Push e to the back of P
33:    u:=other_ver(e, l)
34:    if curr_cost + m(u) > c_max or (u = t and curr_cost < c_min) then
35:        Stop the SAT solver and have it return UNKNOWN
36:    if u = t then
37:        Stop the SAT solver and have it return SAT
38:    return u
```

Algorithm 2. Grid-Aware Strategies

```
1: function GetUnassignedEdge(Direction d)
2:     if e = (l, u) or e = (u, l) ∈ E, such that u is to the d from v, exists and
       unassigned then
3:         return e
4:     else
5:         return ∅
6: function ChooseDirOrdered
7:     for all d ∈ D do
8:         if GetUnassignedEdge(d) ≠ ∅ then
9:             return GetUnassignedEdge(d)
10:    return ∅
11: function GridAwareNextEdge
12:    if stage = init then
13:        if ChooseDirOrdered({south, west}) ≠ ∅ then
14:            return ChooseDirOrdered(south, west)
15:        stage:=spend
16:    if stage = spend then
17:        if l_x = t_x then
18:            stage:=sec_init
19:            sec_init_main_dir:= s to the south of t ? south : north
20:        else
21:            d:= ChooseDirOrdered({north, south, east, west})
22:            if d = west and there is no simple path from l to t then
23:                Stop the SAT solver and have it return unknown
24:            return d
25:    if stage = sec_init then
26:        if ChooseDirOrdered({sec_init_main_dir, east}) ≠ ∅ then
27:            return ChooseDirOrdered({sec_init_main_dir, east})
28:        stage:=sec_spend
29:    if stage = sec_spend then
30:        d:= ChooseDirOrdered({north, south, west, east})
31:        if d = east and there is no simple path from l to t then
32:            Stop the SAT solver and have it return unknown
33:        return d
```

1. Solve: the main function invoked by the user.
2. OnDecision: this function is invoked by the underlying SAT solver to get a decision literal when it has to take a decision.
3. OnImplication: invoked by the SAT solver whenever it derives a new implication (that is, whenever a value for a variable is forced by propagation).
4. OnBacktrack: invoked by the SAT solver whenever it backtracks.
5. PathPushBack: a multi-functional auxiliary function, explained later.

Solve receives the graph G, the source s, the target t, the minimal cost c_{min} and the maximal cost c_{max}. It returns a path P from s to t, whose cost is bounded by $[c_{min}, c_{max}]$, if available. The function starts at line 2 by computing

1. Connectivity variables
   (a) Boolean a_e: a_e is 1 iff $e \in E$ is active.
   (b) Boolean a_v: a_v is 1 iff $v \in V$ is active.
2. Cost variables
   (a) BV c_v: the cost of the path from s to v
   (b) Boolean dir_e: the direction of $e \in E$
3. Connectivity constraints
   (a) a_e implies a_v and a_u, where $e = (v, u)$
   (b) Each vertex v has exactly n active neighbor edges, where:
      i. n=0 if the vertex is inactive
      ii. n=1 if v is the source or the target
      iii. n=2 if v is an active internal vertex
4. Cost constraints
   (a) The active edge touching the source s is s-outgoing
   (b) The active edge touching the target t is t-incoming
   (c) For every active internal vertex v, there must be one v-outgoing and one v-incoming active edge
   (d) $c_s = 0$
   (e) $c_v = c_e + c_u$, given an active internal vertex v, where e, touching v and u, is the v-incoming edge
   (f) $c_{min} \leq c_t \leq c_{max}$

Fig. 2. Translating bounded-path to BV

the minimal cost $m(v)$ from each node v to the target t with one invocation of the Dijkstra algorithm. As we will see, the minimal costs are required for the decision strategies and conflict analysis. At line 3, connectivity constraints are generated and bit-blasted to SAT (word-level preprocessing can also be applied before bit-blasting). The SAT solver is not yet invoked at this stage. At line 4, the target cost tc, comprising the middle of the range $[c_{min}, c_{max}]$, is computed. The algorithm will try to build a path from s to t whose cost is as close as possible to tc (in accordance with the actual skew minimization requirement).

The main loop of the algorithm starts at line 6. It uses the following variables, initialized at line 5:

1. P holds the edges of a simple path starting at s. If the algorithm completes successfully, P will hold a path from s to t bounded by $[c_{min}, c_{max}]$.
2. l contains the latest vertex of the generated path from s to t.
3. *curr_cost* contains the overall cost of (the edges of) P so far.
4. *stage* contains the current stage of the decision strategy (explained later in Sects. 5.2 and 5.3).

The main loop invokes the SAT solver at line 7. The solver may return three possible results. If the solver returns SAT, then P is guaranteed to contain a bounded path from s to t; thus P is returned to the user. If it returns UNSAT, there is no solution, and a special value is returned to the user.

In addition, the solver may return the value UNKNOWN, meaning that a graph conflict was encountered and refinement is required. A *graph conflict* is a situation where no path from s to t with prefix P and cost bounded by $[c_{min}, c_{max}]$ exists. Our algorithm may identify three types of graph conflicts, shown in Fig. 3 and discussed later. When a graph conflict is encountered, the algorithm refines the problem by adding a new *graph conflict clause* which prevents regeneration of the current path P. The graph conflict clause contains activation variables corresponding to the edges in the path P, negated (the clause can be optionally minimized by removing edges from its tail as long as the conflict still occurs). Then the algorithm continues to the next iteration of the loop. After restarting (where by restarting we mean backtracking to decision level 0), the algorithm will pick the same decisions until but not including the latest edge, which has to be different in order to satisfy the graph conflict clause.

5.2 Interactive SAT Solving

We continue the presentation of Algorithm 1. Given a vertex v, let $nbors(v)$ be the set of v's neighbors (recall that v's neighbors are the edges touching v). Given a vertex v and an edge $e \in nbors(v)$, the *other vertex* of e, $other\_ver(e, v)$, is the vertex $u \neq v$, touched by e.

Consider the function OnDecision, invoked by the SAT solver to pick the next decision. It receives the current decision level d and returns an unassigned literal, which is picked by the SAT solver as the next decision literal.

At each stage of the algorithm, let the *cost low bound (CLB)* be $c(P) + m(l)$, that is, the cost of the current path P from s to the latest vertex l plus the pre-computed minimal cost from l to t. Once CLB is greater than or equal to the target cost, the algorithm enters the *shortest path stage* `shortestp` (see lines 15 to 16), where the cost of any path from s to t with prefix P cannot be lower than the target cost. Hence, the algorithm picks an edge so as to have CLB as low as possible after the edge is picked (lines 17 to 19). Note that if the pre-computed shortest path is still not occupied, the algorithm will arrive at t, where the path cost is exactly the target cost. If the shortest path stage is not entered, OnDecision invokes a core decision strategy (described in Sect. 5.3) to pick the next unassigned decision literal. The choice is crucial for performance, but does not alter the correctness.

After an edge is picked, OnDecision invokes the auxiliary function PathPushBack, providing it the edge e and the decision level d. Normally, PathPushBack pushes e to the end of P and returns the new latest vertex l, and then OnDecision returns a_e as the next decision literal (all this is unless PathPushBack discovers a graph conflict or finds that the problem is satisfied). We will get back to the functionality of PathPushBack a bit later.

The function OnImplication is invoked by the SAT solver whenever its Boolean Constraint Propagation (BCP) learns a new implication. It receives the implied literal. If the literal activates an edge e touching the latest vertex l, then e is pushed to P using PathPushBack and l is updated accordingly.

Now consider PathPushBack. First, if the function is invoked when a new decision is taken, it creates a backtrack point at decision level d (lines 29 to 30) so as to let the algorithm (or, more specifically, function OnBacktrack) restore all the relevant variables when (and if) the SAT solver backtracks to decision level d. Creating the backtrack point and backtracking whenever required is essential to maintaining the consistency of the algorithm. Then PathPushBack updates the current cost *curr_cost* and pushes e to the end of P.

Line 34 of PathPushBack checks conditions 1 and 2 in Fig. 3 that might trigger a graph conflict (condition 3 is discussed in Sect. 5.3). First, a graph conflict occurs when the target t is reached, but the cost is not bounded. Note that triggering a graph conflict on this occasion is essential to guaranteeing the soundness of the algorithm. Second, a graph conflict is identified when CLB exceeds the maximal value c_{max} for any non-target vertex. This is not necessary for soundness, but advisable for pruning the search space, thus improving performance. If a graph conflict is identified, PathPushBack stops the SAT solver and asks it to return UNKNOWN.

If no graph conflict is identified, the algorithm checks whether the target is reached within the required cost, in which case it stops the SAT solver and has it return SAT. Finally, if none of the stopping conditions were triggered, PathPushBack returns the new latest vertex on the path.

1. P connects s to t, but P's cost is not within $[c_{min}, c_{max}]$
2. The CLB $c(P) + m(l)$ exceeds the maximal value c_{max} and $l \neq t$
3. The target t is no longer reachable (see an example in Fig. 4a)

Fig. 3. Graph conflict conditions

5.3 Core Decision Strategies

This section proposes the core decision strategies for Algorithm 1. We start by proposing the following simple *graph-aware* strategy, applicable to finding a bounded path in any graph: *go away from the target until the shortest path stage is entered*. This is done by always preferring an edge e such that CLB, after picking e, is the lowest possible. Unfortunately, this simple strategy cannot be used for MCR in our setting, since it ignores the track minimization requirement.

Recall the grid graph related definitions from Sect. 2. We propose a *grid-aware* decision strategy for the problem of finding a bounded path from $s = (s_x, s_y)$ to $t = (t_x, t_y)$ in a grid graph G, where the maximal x-coordinate/y-coordinate is X/Y, respectively. We make the following assumptions regarding the input problem without restricting the generality: (a) G is mainly vertical; (b) $s_y < t_y$ or ($s_y = t_y$ and $|s_y| \leq |Y - s_y|$); (c) $s_x \leq t_x$. Any grid graph can be transformed to meet these conditions by rotating G by 90°, if necessary, to meet the first condition, and choosing the point (0,0) out of the 4 corners to meet the last two conditions.

Our grid-aware strategy is designed to find a bounded path in a grid graph, keeping two main goals in mind:

1. *Graph conflicts-awareness*: try to avoid graph-aware conflicts and identify them when they cannot be avoided.
2. *Track minimization*: try to minimize the number of tracks in the path.

The function GridAwareNextEdge in Algorithm 2 implements the strategy (it is intended to be called at line 20 in Algorithm 1). The algorithm has five stages, where the shortest path stage `shortestp` is entered whenever CLB is greater than or equal to the target cost at any other stage as discussed in Sect. 5.2 and shown in Algorithm 1. The remaining four stages are explained below.

During the initial stage `init`, the algorithm goes towards the corner $(0, 0)$, that is, southwards and westwards, whenever possible. See Fig. 4a for an illustration and lines 12 to 15 in Algorithm 2 for the implementation of stage `init`. The implementation applies an auxiliary function ChooseDirOrdered, which receives an ordered sequence of directions $D = \{d_1, d_2, \ldots\}$. It returns an unassigned edge e touching l, such that $other\_ver(e, l)$ is to the d_i of l, where i is the lowest possible index, such that e exists and is unassigned. After the `init` stage, the algorithm enters the `spend` stage.

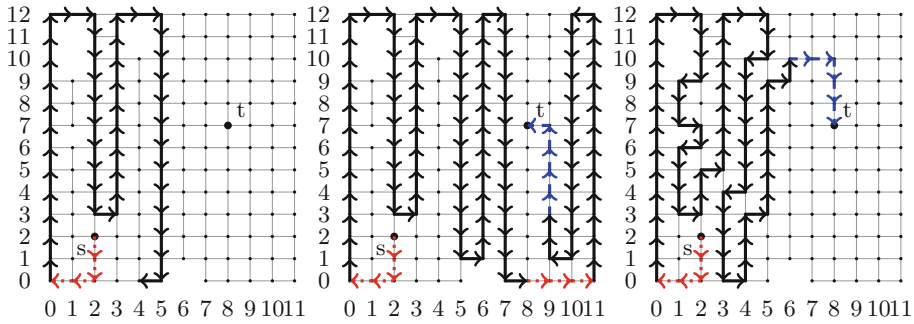
During the `spend` stage, the algorithm tries to "spend the cost" using as few tracks as possible by moving along the vertical tracks coast-to-coast whenever possible (recall that the vertical tracks have more edges than the horizontal tracks by our convention). When moving along a vertical track is no longer possible, the algorithm turns towards the target t (in order not to block the way to t). This stage can finish with the following possible outcomes:

1. The algorithm is turned away to the west by the SAT solver's propagation and there is no longer any path from l to t (line 22), where the latter condition is checked using DFS. In this case, a graph conflict corresponding to condition 3 in Fig. 3 is triggered, and the algorithm stops the SAT solver. An example of such an outcome is shown in Fig. 4a. In this case, a conflict clause is generated by Algorithm 1. After restarting the SAT solver, the algorithm follows the same path as before until an implication in the new conflict clause turns it to the east and the graph conflict is avoided, as shown in Fig. 4b.
2. PathPushBack in Algorithm 1 halts the main loop of Algorithm 1 due to a graph conflict or when a bounded path from s to t is found (the latter is an unlikely corner case).
3. The shortest path stage `shortestp` is entered.
4. The vertical track of t is reached (line 17), in which case the second initialization stage `sec_init` is entered.

During the second initial stage `sec_init` (lines 25 to 28), the algorithm goes to one of the eastern corners according to the relative position of l with respect

to t. When moving to the corner is no longer possible, the algorithm enters the second spend stage `sec_spend`.

During the second spend stage `sec_spend` (lines 29 to 33), the algorithm spends the cost similarly to the first spend stage `spend`, except that it moves eastwards and does not stop when the vertical track of t is reached. In our example in Fig. 4b, stage `sec_spend` is finished when CLB becomes equal to the target cost and the shortest path stage is entered.



(a) Conflict (b) Solution (c) Packing

Fig. 4. Application of the grid-aware strategies. Assume the cost of each edge is 1 and the requested cost range is $[70, 80]$. The red dotted edges correspond to the initial stages `init` and `sec_init`, the black solid edges correspond to the cost spend stages `spend` and `sec_spend`, while the blue dashed edges correspond to the shortest path stage `shortestp`. A graph conflict situation is shown in Fig. 4a; the eventual solution after the conflict is handled is shown in Fig. 4b; the packing effect is shown in Fig. 4c.

Remark 1. Assume the grid-aware strategy can go either eastwards or along the vertical track during the `spend` stage after circumventing an obstacle. Consider the choices at vertex $(2, 9)$ in Fig. 4a for an example. Algorithm 2 prefers continuing along the vertical track. An alternative would be preferring to go westwards (implementation-wise, that would require replacing the parameters to ChooseDirOrdered at line 21 in Algorithm 2 by $\{$`west`, `north`, `south`, `east`$\}$). Similarly, such an algorithm would prefer going eastwards whenever possible during the `sec_spend` stage. We call this alternative approach *packing*. Its impact is shown in Fig. 4c. Packing is designed to use all the available space in the grid graph, thus it is better suited to cases where there are many obstacles or the target cost is high. However, it comes at the price of excessive track usage. Note the "ripple effect" of occupying the horizontal tracks 6,5 and 4, created by the turn westwards at point $(2, 6)$.

Remark 2. Our approach is expected to generate non-optimal results in terms of track minimization for a generic rectilinear polygon as compared to a (possibly

holed) mainly vertical rectangle, since some of the polygon's rectangles might be mainly horizontal (even though, *most* of the edges are vertical). We leave designing an adaptive strategy that would change the explored dimension on-the-fly to future work.

6 Experimental Results

First, we present experiments conducted on artificially generated test benchmarks. The benchmarks and detailed results are available in [11]. The benchmarks comprise diversified parametrized instances of bounded-path in grid graphs, generated as follows:

1: **for all** $t \in \{10^1, 10^2, 10^3\}$ **do**
2: **for all** $d \in \{0, 0.25, 0.5, 0.75, 1\}$ **do**
3: **for all** $vcost \in \{102, 104, 106, 108, 110, 112, 114, 116, 118, 120\}$ **do**
4: **for all** $r \in \{0.1, 0.2, 0.3, 0.4, 0.5\}$ **do**
5: $c{:=}S \times r$ ▷ S is the overall edges cost
6: Generate a square grid of size $t \times t$ with randomly set source and target. Remove any node v (along with the edges $nbors(v)$) with probability d/t. Set the cost of each horizontal and vertical edge to 100 and $vcost$, respectively. Set the target cost to c and the allowed skew to 2.5 %.

The parameters were selected as follows so as to diversify the instances and to be able to analyze various aspects of the algorithms' performance: (a) t stands for the number of tracks along each dimension, hence $t \times t$ is the grid size; (b) d determines the dilution rate. We remove $(d/t)t^2 = dt$ vertices on average at random, so as to defragment the grid graph. (c) $vcost$ determines the vertical cost, while the horizontal cost is static; (d) r determines the target cost as a function of the overall edges cost S.

We compared the following algorithms, implemented on top of Intel's eager SMT solver Hazel: (a) `BV`: reduction to BV, described in Sect. 4. (b) `Graph`: Algorithm 1 with the graph-aware strategy described in the first paragraph of Sect. 5.3. (c) `Grid`: Algorithm 1 with the grid-aware strategy in Algorithm 2 (d) `GridP`: Algorithm 1 with the grid-aware strategy Algorithm 2 and packing (recall Remark 1 in Sect. 5.3).

We used machines with 32Gb of memory running Intel® Xeon® processors with 3Ghz CPU frequency. The time-out was set to 600 sec.

Table 1 presents the number of instances solved within the time-out per grid size. Table 2 shows the overall number of tracks used for all the algorithms (except `BV`) on benchmarks solved by all these algorithms. Tables 3 and 4 show the number of instances `Grid` and `GridP`, respectively, solve per each combination of r and d values for $s = 10^2$. The main conclusions are as follows.

Plain translation to BV does not scale even to 100×100 grids. To validate that this result is independent of the underlying solver, in an additional experiment, we verified that the two leading SAT solvers Lingeling [5,6] and Glucose 4.0 [1,2] can solve none of the CNF instances corresponding to benchmarks with $t = 100$ and $d \in \{0, 1\}$. The CNF instances, available in [11], were dumped by Hazel

Table 1. Solved per grid size

s	#	BV	Graph	GridP	Grid
10	250	138	217	223	200
100	250	0	171	226	151
1000	250	0	158	194	154
Sum	750	138	546	643	505

Table 2. Tracks used in %

s	Graph	GridP	Grid
10	58	56	45
100	44	59	22
1000	43	51	21

Table 3. `Grid`: solved out of 10 instances per cell, given $r\&d$ for $s = 10^2$

r/d	0	0.25	0.5	0.75	1
0.1	10	9	9	8	10
0.2	10	10	8	7	7
0.3	10	9	8	6	2
0.4	10	8	0	0	0
0.5	10	0	0	0	0

Table 4. `GridP`: solved out of 10 instances per cell, given $r\&d$ for $s = 10^2$

r/d	0	0.25	0.5	0.75	1
0.1	10	10	10	10	10
0.2	10	10	10	10	10
0.3	10	10	10	10	10
0.4	10	10	10	10	10
0.5	10	7	5	3	1

after the world-level preprocessing stage. We could not run external BV solvers as is, since they do not support conditional cardinality constraints.

`GridP` is the most robust strategy as it solves the most test instances. Moreover, when the target cost is not too high ($r < 0.5$), `GridP` solves all the instances for $s = 10^2$. `Grid` cannot solve instances with high target costs and/or dilution rates. Hence, as expected, packing is useful for handling grids with many obstacles. The performance of `Graph` is surprisingly good for such a simple strategy.

As expected, `Grid` is by far the best algorithm in terms of track minimization.

We also conducted experiments on a family of real-world instances generated by Intel's clock routing flow. The family has 51 benchmarks. The number of edges in the benchmarks ranges from 70,492 to 4,436,948, with an average of 1,203,631, while the number of vertices ranges between 44,320 and 2,837,800 with the average of 780,782. The results can be summarized as follows: (a) `BV` solved none of the 51 instances, `Grid` and `Graph` solved all the instances, while `GridP` solved 49 instances. (b) `Grid` used 285 tracks overall, `Graph` used 863 tracks, while `GridP` used 387 tracks. Hence, unlike in the case of randomized test instances, it pays to use `Grid` on real-world instances. `Grid` is successfully applied for clock routing automation at Intel.

7 Conclusion

We have presented an SMT-based approach to automating the matching constrained routing problem that has emerged in the clock routing of integrated circuits. We reduced the problem to bounded-path, that is, the problem of finding a simple path, whose cost is bounded by a user-given range, connecting two given vertices in an undirected positively weighted graph. We have shown that bounded-path can be solved by applying an eager bit-vector solver, but only if the solver is enhanced with a dedicated graph-aware decision strategy and

graph-aware conflict analysis. Our solution scales to graphs having millions of edges and vertices. It has been successfully deployed at Intel as part of the core engine for automatic clock routing.

Acknowledgments. We are grateful to Nachum Dershowitz for suggesting and proving that bounded-path is NP-hard (the paper's proof differs from Nachum's proof). We thank Paul Inbar, Eran Talmor, and Vadim Ryvchin for their useful comments.

References

1. Audemard, G., Simon, L.: The Glucose SAT Solver. http://www.labri.fr/perso/lsimon/glucose/. Accessed: 05–January–2015
2. Audemard, G., Simon, L.: GLUCOSE 2.1: aggressive, but reactive, clause database management, dynamic restarts (system description). In: Pragmatics of SAT 2012 (POS 2012), June 2012
3. Barrett, C., Donham, J.: Combining SAT methods with non-clausal decision heuristics. Electr. Notes Theor. Comput. Sci. **125**(3), 3–12 (2005)
4. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB Standard: version 2.0. In: Gupta, A., Kroening, D., (eds.) Proceedings of the 8th International Workshop on Satisfiability Modulo Theories, Edinburgh (2010)
5. Biere, A.: Lingeling, Plingeling and Treengeling. http://fmv.jku.at/lingeling/. Accessed: 05–January–2015
6. Biere, A.: Lingeling, plingeling and treengeling entering the SAT competition. In: Proceedings of SAT Competition 2013, p. 51 (2013)
7. Biere, A., Le Berre, D., Lonca, E., Manthey, N.: Detecting cardinality constraints in CNF. In: Sinz, C., Egly, U. (eds.) SAT 2014. LNCS, vol. 8561, pp. 285–301. Springer, Heidelberg (2014)
8. Biere, A., Heule, M., Van Maaren, H., Walsh, T.: Handbook of Satisfiability, volume 185 of Frontiers in Artificial Intelligence and Applications. IOS Press, Amsterdam (2009)
9. Brummayer, R., Biere, A.: Boolector: An efficient SMT solver for bit-vectors and arrays. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 174–177. Springer, Heidelberg (2009)
10. Cimatti, A., Franzén, A., Griggio, A., Sebastiani, R., Stenico, C.: Satisfiability modulo the theory of costs: foundations and applications. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 99–113. Springer, Heidelberg (2010)
11. Erez, A., Nadel, A.: Finding bounded path in graph using SMT for automatic clock routing: Benchmarks and detailed results. http://goo.gl/sM4xxz
12. Amit Erez and Alexander Nadel. Finding bounded path in graph using SMT for automatic clock routing: Extended version. https://goo.gl/jtq669
13. Fishburn, J.P.: Clock skew optimization. IEEE Trans. Comput. **39**(7), 945–951 (1990)
14. Ganesh, V., Dill, D.L.: A decision procedure for bit-vectors and arrays. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 519–531. Springer, Heidelberg (2007)
15. Ganzinger, H., Hagen, G., Nieuwenhuis, R., Oliveras, A., Tinelli, C.: DPLL(T): fast decision procedures. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 175–188. Springer, Heidelberg (2004)

16. Gao, Q., Yao, H., Zhou, Q., Cai, Y.: A novel detailed routing algorithm with exact matching constraint for analog and mixed signal circuits. In: Proceedings of the 12th International Symposium on Quality Electronic Design (ISQED 2011), pp. 36–41. IEEE, Santa Clara, 14–16 March 2011
17. Hadarean, L.: An Efficient and Trustworthy Theory Solver for Bit-vectors in Satisfiability Modulo Theories. dissertation, New York University (2015)
18. Ioannidou, K., Nikolopoulos, S.D.: The longest path problem is polynomial on cocomparability graphs. Algorithmica **65**(1), 177–205 (2013)
19. Itai, A., Papadimitriou, C.H., Szwarcfiter, J.L.: Hamilton paths in grid graphs. SIAM J. Comput. **11**(4), 676–686 (1982)
20. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Chaff, S.M.: Engineering an efficient SAT solver. In: Proceedings of the 38th Design Automation Conference (DAC 2001), pp. 530–535. ACM, Las Vegas, 18–22 June 2001
21. Nadel, A.: Bit-vector rewriting with automatic rule generation. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 663–679. Springer, Heidelberg (2014)
22. Ozdal, M.M., Hentschke, R.F.: Maze routing algorithms with exact matching constraints for analog and mixed signal designs. In: 2012 IEEE/ACM International Conference on Computer-Aided Design (ICCAD 2012), pp. 130–136. IEEE, San Jose, 5–8 November 2012
23. Ryzhenko, N., Burns, S.: Standard cell routing via boolean satisfiability. In: Groeneveld, P., Sciuto, D., Hassoun, S., (eds.) The 49th Annual Design Automation Conference DAC 2012, pp. 603–612. ACM, San Francisco, 3–7 June 2012
24. Sabharwal, A.: Symchaff: a structure-aware satisfiability solver. In: Veloso, M.M., Kambhampati, S., (eds.) Proceedings, The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference, pp. 467–474. AAAI Press / The MIT Press, Pittsburgh, 9–13 July 2005
25. Sherwani, N.A.: Algorithms for VLSI Physical Design Automation, Chap. 8, 3rd edn. Kluwer, The Netherlands (1998)
26. Sherwani, N.A.: Algorithms for VLSI Physical Design Automation, Chap. 11. Kluwer, The Netherlands (1988)
27. Marques Silva, J.P., Sakallah, K.A.: GRASP: a search algorithm for propositional satisfiability. IEEE Trans. Comput. **48**(5), 506–521 (1999)
28. Surynek, P.: A simple approach to solving cooperative path-finding as propositional satisfiability works well. In: Pham, D.-N., Park, S.-B. (eds.) PRICAI 2014. LNCS, vol. 8862, pp. 827–833. Springer, Heidelberg (2014)
29. Umans, C., Lenhart, W.: Hamiltonian cycles in solid grid graphs. In: Proceedings of the 38th Annual Symposium on Foundations of Computer Science 1997, pp. 496–505, October 1997
30. Wood, R.G., Rutenbar, R.A.: FPGA routing and routability estimation via Boolean satisfiability. pp. 222–231 (1998)
31. Wu, G., Jia, S., Wang, Y., Zhang, G.: An efficient clock tree synthesis method in physical design. In: IEEE International Conference of Electron Devices and Solid-State Circuits (EDSSC 2009), pp. 190–193, December 2009