Mathematical Programming
https://doi.org/10.1007/s10107-023-01962-4

FULL LENGTH PAPER

Series B

Faster goal-oriented shortest path search for bulk and incremental detailed routing

Markus Ahrens[1] · **Dorothee Henke**[2] · **Stefan Rabenstein**[3] · **Jens Vygen**[3]

Received: 11 July 2022 / Accepted: 12 April 2023

Abstract
We develop new algorithmic techniques for VLSI detailed routing. First, we improve the goal-oriented version of Dijkstra's algorithm to find shortest paths in huge incomplete grid graphs with edge costs depending on the direction and the layer, and possibly on rectangular regions. We devise estimates of the distance to the targets that offer better trade-offs between running time and quality than previously known methods, leading to an overall speed-up. Second, we combine the advantages of the two classical detailed routing approaches—global shortest path search and track assignment with local corrections—by treating input wires (such as the output of track assignment) as reservations that can be used at a discount by the respective net. We show how to implement this new approach efficiently.

Mathematics Subject Classification 90C90 · 90C27 · 68U05 · 51-08 · 90C35

1 Introduction

The task of VLSI routing [3, 21] is to connect the set of pins of every net on a chip by wires so that wires of different nets are sufficiently far apart and various other

✉ Stefan Rabenstein
rabenstein@dm.uni-bonn.de

Markus Ahrens
markus.johannes.ahrens@ibm.com

Dorothee Henke
dorothee.henke@math.tu-dortmund.de

Jens Vygen
vygen@dm.uni-bonn.de

1 IBM Deutschland Research & Development GmbH, Böblingen, Germany

2 Department of Mathematics, TU Dortmund University, Dortmund, Germany

3 Research Institute for Discrete Mathematics, Hausdorff Center for Mathematics, University of Bonn, Bonn, Germany

Published online: 24 May 2023

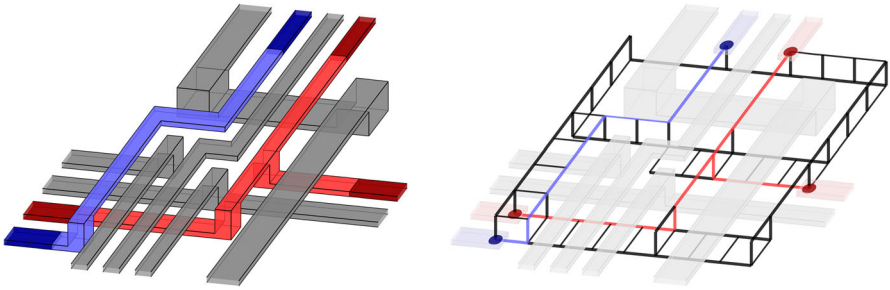

Fig. 1 Left: tiny part of a routed chip with two layers. The blue wires connect the two dark blue pins and the red wires connect the three dark red pins. Pins of the same color belong to the same net. The gray wires are part of the connections of multiple other nets with pins outside of the visible region. Right: the relevant part of the detailed routing graph before routing the blue and the red net. Steiner trees for the red net and the blue net (color figure online)

constraints are met. See Fig. 1(left) for an example. In a simple but useful model, we have a huge 3-dimensional grid graph (the *detailed routing graph*), and the *pins* are vertices in this graph. Each *net* is a set of pins and needs to be connected by a Steiner tree in the detailed routing graph. The Steiner trees of distinct nets must be vertex-disjoint. The detailed routing graph is induced by *routing tracks*, which are pre-computed parallel lines on each layer. Every two routing tracks on adjacent layers are orthogonal to each other and induce one vertex on each of these tracks. These two vertices are connected by an edge; metal connections along those edges (connecting adjacent layers) are called *vias*. Depending on the manufacturing process, vertices on adjacent tracks of the same layer may also be connected by an edge.

Typically, one first computes a *global routing*, a rough packing of wires that ignores all local constraints but guarantees that the wires in certain areas do not require more space than available. Global routing allows for globally optimizing objectives such as power consumption and timing constraints [9, 15].

The output of global routing then restricts the search space for every net in detailed routing, where many complicated rules need to be obeyed and one essentially routes one net at a time. While the detailed routing graph formed by routing tracks on an entire chip can contain about 10^{13} vertices on 10–20 layers, the restricted area corresponding to the global routing solution for a net results in a much smaller detailed routing graph, with rarely more than 10^8 vertices. Nevertheless, these subgraphs are still huge, and there are millions of nets to connect. Two general strategies have been proposed (cf. [21]).

The first approach is based on a fast subroutine to find a shortest path that connects two metal components, each of which can consist of a pin or a set of previously computed wires connecting a subset of the pins of that net. The subgraph is given by the global routing solution, excluding vertices and edges that would result in a conflict to previously routed wires. For an example of the resulting graph, see Fig. 1(right). To allow for an efficient packing of wires and to model various aspects such as signal delays, one uses different costs for horizontal and for vertical edges on each layer as well as for vias connecting two adjacent layers.

The second approach first considers the layers one after the other and assigns wires to routing tracks so that the most important detailed routing rules are satisfied, at least for most wires. This is often called *track assignment* [4, 20]. Then detailed routing tries to correct violations locally. A very similar detailed routing problem occurs when a detailed routing has already been computed, but a few changes to the input have been made (for example corrections of the logical behavior or to speed up signals that arrived too late). In both cases, one asks for an *incremental* detailed routing, largely following the input but deviating where necessary. However, local corrections are often not possible if the routing is very dense.

Goal-oriented path search (sometimes called A^*) is a classical speed-up technique of Dijkstra's shortest path algorithm [5]. It is based on a feasible potential that estimates the distance to the targets [8, 13, 18]. Instead of the undirected graph with the original edge cost $c(e)$, we orient each edge in both ways and run Dijkstra's algorithm with the *reduced cost* $c_\pi(e) := c(e) - \pi(v) + \pi(w)$ for every edge e directed from v to w, where the vertex potentials π are chosen so that c_π is nonnegative and $\pi(t) = 0$ for every target t. These conditions imply that $\pi(v)$ is a lower bound on the distance between v and the closest target. The better this lower bound is, the fewer vertices this goal-oriented version of Dijkstra's algorithm must label before it knows a shortest path to a target.

Hence, there is a trade-off between a possible preprocessing time, the query time to compute the potential of a vertex, and the quality of the lower bound. For example, in subgraphs of unweighted grid graphs, the ℓ_1-distance to the nearest target can be a reasonable choice for π [10]. A better estimate, which however requires substantial preprocessing, was suggested by [16]. In this paper, we propose new methods with better trade-offs than previously known.

Moreover, we combine the advantages of the two classical detailed routing approaches mentioned above. Our new, more global approach treats given input wires (e.g., the output of track assignment) as so-called *reservations*. A reservation for a net N is a set of edges reserved for N until N is routed: no other net must use these edges. We encourage, but not force, the detailed router to follow the reservations where feasible. This is achieved by finding a shortest path where reservations of the currently routed net can be used at a discount (so we reduce the cost of reserved edges by a fixed factor smaller than 1).

However, this does not work well together with the classical goal-oriented techniques. For example, if there are some reservations (edges) that can be used at a 50 % discount, the ℓ_1-distance would have to be divided by 2 in order to induce a feasible potential. This would often be a very inaccurate estimate, leading to an increased number of labels in Dijkstra's algorithm and hence larger running time. We show that our better potentials make goal-oriented Dijkstra not only as fast as without reservations, but in fact faster. Overall, this yields a new efficient incremental detailed routing algorithm.

1.1 Problem statement

Our core problem will consist of computing distances in a weighted grid graph with a simple structure. To define the grid graph, we number the layers $1, \ldots, l$ and let $V = \mathbb{Z} \times \mathbb{Z} \times \{1, \ldots, l\}$ and

$$E = \left\{ \{(x, y, z), (x', y', z')\} \in \binom{V}{2} \mid |x - x'| + |y - y'| + |z - z'| = 1 \right\}$$

be the vertex set and edge set of an infinite grid with l layers. Edges connecting adjacent layers are called *vias*, edges in x-direction are *horizontal* and edges in y-direction *vertical*. We will consider finite subgraphs of $G = (V, E)$. These subgraphs correspond to the area defined by the global routing solution and to the restriction to the routing tracks that can be used for the current net. Often, many vertices of these subgraphs will have degree 2 and will not be considered explicitly, but we ignore this here for the sake of a simpler exposition.

Every layer has a preference direction ($\leftrightarrow$ or $\updownarrow$, the direction of its tracks); edges in the other direction are more expensive or sometimes even forbidden, depending on the manufacturing process. Horizontal and vertical layers alternate. Moreover, the layers have very different electrical properties, which is reflected by appropriate edge costs. In the simplest model, the cost of an edge depends only on its direction and the layer: let $c_z^{\leftrightarrow}, c_z^{\updownarrow} > 0$ for $z \in \{1, \ldots, l\}$ and $c_{z,z+1} > 0$ for $z \in \{1, \ldots, l-1\}$; then

$$c(\{(x, y, z), (x', y', z')\}) = \begin{cases} c_z^{\leftrightarrow} & \text{if } x' = x + 1 \\ c_z^{\updownarrow} & \text{if } y' = y + 1 \\ c_{z,z'} & \text{if } z' = z + 1 \end{cases}.$$

In a more general model, a rectilinear grid induces rectangular regions, called *tiles*, and the cost also depends on the tile. Let

$$-\infty = \xi^0 < \xi^1 \leq \ldots \leq \xi^p < \xi^{p+1} = \infty,$$
$$-\infty = \upsilon^0 < \upsilon^1 \leq \cdots \leq \upsilon^q < \upsilon^{q+1} = \infty$$

be integer coordinates that define the rectangular tiles

$$V_z^{ij} = \left\{ (x, y, z) \in V \mid \xi^i \leq x \leq \xi^{i+1},\ \upsilon^j \leq y \leq \upsilon^{j+1} \right\},$$

and set

$$E_z^{ij} = \left\{ \{(x, y, z), (x', y', z')\} \in E \mid \xi^i \leq x \leq x' \leq \xi^{i+1},\ \upsilon^j \leq y \leq y' \leq \upsilon^{j+1},\ z \leq z' \right\}.$$

Now we have costs $c_z^{ij\leftrightarrow}, c_z^{ij\updownarrow}, c_{z,z+1}^{ij} > 0$ that also depend on the tile and define the edge costs accordingly. If an edge belongs to more than one tile, the minimum cost applies. See Fig. 2 for an example. We allow that two (but not three) consecutive

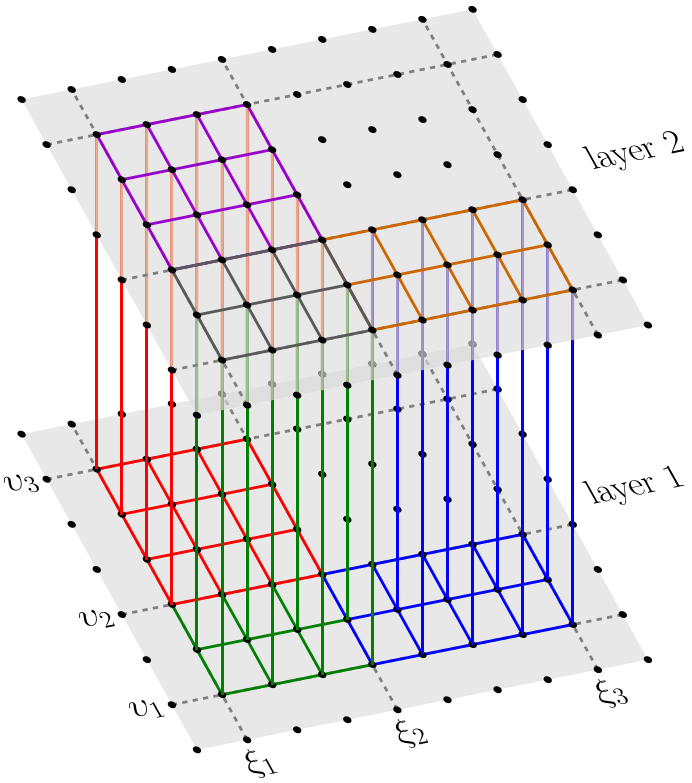

Fig. 2 Example of the general cost model with $p = 3$, $q = 3$, and $l = 2$. The red edges form the set E_1^{12}. The blue horizontal edges have cost $c_1^{21\leftrightarrow}$, the blue vertical edges have cost $c_1^{21\updownarrow}$, and the blue via edges have cost $c_{1,2}^{21}$. The cost of the blue vertical edges on ξ_2 is given by $\min\{c_1^{11\updownarrow}, c_1^{21\updownarrow}\}$, which is $c_1^{21\updownarrow}$ in this example. The cost of the green via edges on ξ_2 is $\min\{c_{1,2}^{11}, c_{1,2}^{21}\}$, which is $c_{1,2}^{11}$ here. The edges that are not drawn have cost infinity because, in this example, they lie outside the area corresponding to the global routing solution (color figure online)

coordinates are identical, i.e., $\xi^i = \xi^{i+1}$ or $\upsilon^j = \upsilon^{j+1}$, in order to model a cheap cost at one x- or y-coordinate only.

With this more general model, one can, for example, punish wires on low layers near the electrical source of a net (which would lead to poor delays) or implement a discount on reservations as we will describe in detail in Sect. 5.2. Moreover, we can set edge costs to infinity outside the area corresponding to the global routing solution so that the distances in (G, c) reflect necessary detours that are implied by routing in this subgraph.

Given a finite subgraph $G' = (V', E')$ of G and sets $S, T \subseteq V'$, we look for a shortest (minimum-cost) path from S to T in G' with respect to the cost function c. The graph G' does normally not contain vertices and edges whose use would result in a conflict to nets routed previously (an exception will be described at the end of Sect. 5.1), and it can have a very complicated structure. For a goal-oriented path search, we define a potential $\pi(v)$ for every vertex $v \in V'$ by the distance to T in G (instead

of G'):

$$\pi(v) := \mathrm{dist}_{(G,c)}(v, T).$$

The idea is that distances in G are much easier to compute than in the subgraph G' (we will see how fast), but often still give a good lower bound. The reason is that (G, c) has a simple structure, given by the tiles, while G' can be very complicated as it does not contain vertices or edges whose use would result in a conflict to nets routed previously.

This allows us to use Dijkstra's algorithm with the reduced costs c_π in the digraph resulting from G' by orienting every edge in both ways, since the reduced costs are nonnegative. Indeed, we have $c_\pi((v, w)) = c(e) - \pi(v) + \pi(w) = c(e) - \mathrm{dist}_{(G,c)}(v, T) + \mathrm{dist}_{(G,c)}(w, T) \geq 0$ for all $e = \{v, w\} \in E$. After introducing a super-source $\bar{s}$ and arcs $(\bar{s}, s)$ of cost 0 for all $s \in S$, with $\pi(\bar{s}) := \min\{\pi(s) : s \in S\}$, every path P from $\bar{s}$ to T satisfies $c_\pi(P) = c(P) - \pi(\bar{s})$, so shortest $\bar{s}$-T-paths with respect to c_π are shortest $\bar{s}$-T-paths (and correspond to shortest S-T-paths) with respect to c.

The better the lower bound π on the distance to T in (G', c) is, the fewer vertices will be labeled by Dijkstra's algorithm with reduced costs c_π; more precisely all vertices v with $\mathrm{dist}_{(G',c)}(S, v) + \pi(v) < \mathrm{dist}_{(G',c)}(S, T)$ will be processed.

The target set T can consist of a single vertex (corresponding to a pin), but a pin sometimes covers more than one vertex, and when constructing Steiner trees from paths we often want to connect to a connected component that contains wires and more than one pin. We often assume that T is represented as the union of t rectangles, where a rectangle is a vertex set of the form $\{(x, y, z) \in V \mid \xi^- \leq x \leq \xi^+, \upsilon^- \leq y \leq \upsilon^+, z = \zeta\}$ for some $\xi^-, \xi^+, \upsilon^-, \upsilon^+ \in \mathbb{Z}$ and $\zeta \in \{1, \ldots, l\}$. Often t is small in practice. While we can deal with complicated target sets, some of our algorithms work best for simple targets (i.e., small t).

Sometimes it will be useful to assume that this representation is *consistent* with the partition of V into tiles in the following sense: each of the t rectangles representing T fits into the grid, i.e., is of the form $\{(x, y, z) \in V \mid \xi^{i^-} \leq x \leq \xi^{i^+}, \upsilon^{j^-} \leq y \leq \upsilon^{j^+}, z = \zeta\}$ for some indices i^-, i^+, j^- and j^+. This can be achieved by adding at most $2t$ new x-coordinates ξ^i and at most $2t$ new y-coordinates υ^i. We call this procedure *refining the grid with respect to the targets*. See Fig. 3 for an example of the empty grid (i.e., $p = q = 0$) refined with respect to several target rectangles.

1.2 Previous work and our results

In the simple model without regions (i.e., for $p = q = 0$), one can query π (i.e., evaluate $\pi(v)$ for a given query location $v \in V$) easily in $O(tl^2)$ time without preprocessing; see Proposition 2. We show that this can be reduced to $O(tl)$; see Theorem 3. With a preprocessing time polynomial in t and l, we obtain a query time of $O(\log(t + l))$; see Theorem 7. These results will be presented in Sects. 2 and 3.

For the more general model, which is the subject of Sect. 4, Peyer et al. [16] refined the grid with respect to the targets and showed that then the restriction of $\pi : V \to \mathbb{R}_{\geq 0}$

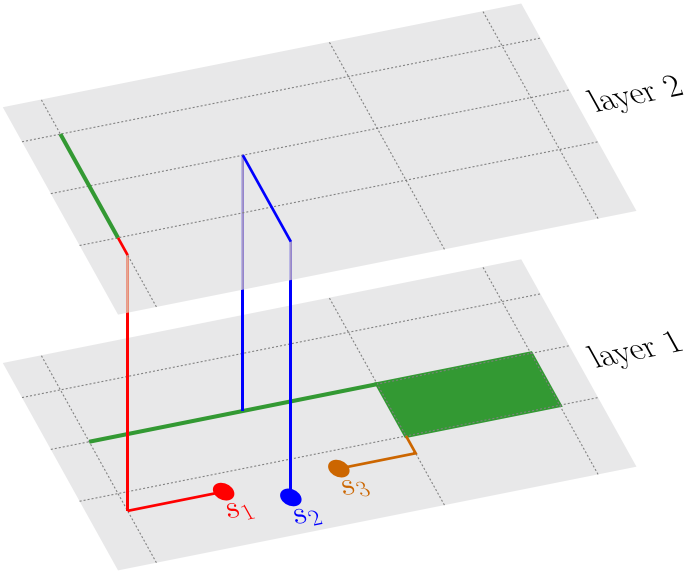

Fig. 3 A target set T consisting of three target rectangles on two layers. The coarsest partition into tiles that is consistent with T is shown by the dotted lines. The figure also shows three possible query locations (s_1, s_2, and s_3) for which we might be interested in the distance to the closest target. The shortest s_1-T-path, the shortest s_2-T-path and the shortest s_3-T-path for a cost function depending only on direction and layer are shown. In this example, the preference directions of layer 1 and 2 are $\leftrightarrow$ and $\updownarrow$, respectively. Nevertheless it is cheapest for the s_3-T-path to stay on layer 1 since its vertical segment is very short

to V_z^{ij} is the minimum of k^2 affine functions for any i, j, z, where k is the number of different horizontal and vertical edge costs, i.e.,

$$k := \left| \left\{ c_z^{ijd} \mid i \in \{0, \ldots, p\}, j \in \{0, \ldots, q\}, d \in \{\leftrightarrow, \updownarrow\}, z \in \{1, \ldots, l\} \right\} \right|. \tag{1}$$

They also showed that all these functions can be computed in $O((p+t)(q+t)lk^4 \log(p+q+l+t))$ time, allowing $O(k^2)$ time queries after this preprocessing (plus $O(\log(p+q+t))$ to find the region containing the given vertex by binary search; here and henceforth p and q refer to the original number of rows and columns, before refining the grid).

We make multiple improvements over the approach of Peyer et al. [16]. By considering domination between affine functions with different slopes, we reduce the number of affine functions that are needed to describe the minimum. By first computing the distances from the edges on the boundaries of the tiles to the targets, we can compute these affine functions faster. Finally, we use a regional query data structure to reduce query time. For any $0 < \epsilon \leq 1$, we can obtain an algorithm with preprocessing time $O((p+t)(q+t)\min\{k, (p+q+1)l\}l^{1+\epsilon}\frac{1}{\epsilon}\log(p+q+l+t))$ and query time $O(\log(p+q+t) + \frac{1}{\epsilon}\log(k+l))$. See Table 1 for an overview.

Our second contribution is a new approach to incremental routing. Rather than trying to correct a given infeasible input routing with local transformations only, we

Table 1 Various methods to compute $\pi(v)$, possibly after preprocessing

Model	Preprocessing time	Query time	Reference
Simple	–	$O(tl^2)$	Proposition 2
Simple	–	$O(tl)$	Theorem 3
Simple	$O(t^2l^3 \log l)$	$O(\log(t+l))$	Theorem 7
General	$O((p+t)(q+t)lk^4 \log(p+q+l+t))$	$O(\log(p+q+t)+k^2)$	[16]
General	$O((p+t)(q+t)\min\{k,(p+q+1)l\}l^{1+\epsilon}$ $\frac{1}{\epsilon}\log(p+q+l+t))$	$O(\log(p+q+t)$ $+\frac{1}{\epsilon}\log(k+l))$	Corollary 15

The running times depend on the number t of target rectangles, the number l of layers, and in the general model on the numbers p and q of coordinates that define the $(p+1)(q+1)$ regions, and on the number k of different horizontal and vertical edge costs (cf. (1)). Note that $k \leq 2(p+1)(q+1)l$

compute a new routing from scratch, at least for all nets for which the input routing does not obey all rules. However, in an incremental routing setting most wires will be legal, i.e., do not have a conflict with any other wire. In order to compute a solution similar to the input where reasonable, we reserve the space occupied by legal input wires for the respective net and allow to use edges corresponding to input wires at a discount. By letting each input wire be a separate tile V_z^{ij}, we can model the discount in the cost function c and work with reduced costs efficiently. When most of the input routing can be used, we can find a shortest path much faster than without a discount.

This makes this new approach not only useful for incremental routing, but also for bulk routing. Treating the output of a track assignment as reservations (wherever it is legal) and then pursuing our new incremental routing approach can combine the advantages of the two classical bulk routing approaches, successive shortest paths and track assignment with local corrections. We explain our new approach in detail in Sect. 5, where we also show experimental results.

2 Distances without preprocessing in the simple model

In the simple model, there is always a shortest path with a very simple structure:

Lemma 1 *Let $c : E \to \mathbb{R}_{>0}$ depend only on direction and layer, and let $r, s \in V$. Then there is a shortest path P between r and s in (G, c) that consists of at most one sequence of horizontal edges, at most one sequence of vertical edges, and hence at most three sequences of vias.*

Proof Let P be a shortest path, and let $P_{[v,w]}$ and $P_{[v',w']}$ be two maximal subpaths of P in the same direction (all-horizontal or all-vertical), say from v to w and from v' to w', respectively, and let $P_{[w,v']}$ be the subpath in between. Suppose, without loss of generality, that $P_{[v,w]}$ and $P_{[v',w']}$ are horizontal paths and that the cost of an edge of $P_{[v,w]}$ is not more expensive than the cost of an edge of $P_{[v',w']}$ (note that these paths may be on different layers). Then translating $P_{[v',w']}$ by adding $w - v'$ to all its vertices, translating $P_{[w,v']}$ by adding $w' - v'$ to all its vertices, and swapping these two paths in P yields a walk from r to s with one maximal horizontal subpath less

and at most the same number of maximal vertical subpaths. Moreover, the cost does not increase. If the walk is not a path, we can shortcut it to a path. By induction, the assertion follows. □

Hence, in order to compute a shortest path, we can enumerate the layers on which the horizontal sequence and the vertical sequence are, and which of the two comes first:

Proposition 2 *Let $c : E \to \mathbb{R}_{>0}$ depend only on direction and layer. Then, without preprocessing, one can compute $\mathrm{dist}_{(G,c)}(s, T)$ for any given $s \in V$ and given $T \subseteq V$ consisting of t rectangles in $O(tl^2)$ time.*

Proof Enumerate over all t rectangles that T is composed of. For each such rectangle R, we can determine the vertex $r \in R$ that is closest to s (geometrically) in constant time. Then, for each pair of layers $z^{\leftrightarrow}, z^{\updownarrow} \in \{1, \ldots, l\}$, we consider two paths that connect r and s. The first one is composed of the path of vias that goes from r to layer $z^{\leftrightarrow}$, followed by the horizontal path that goes to the x-coordinate of s, followed by the path of vias that goes to layer $z^{\updownarrow}$, followed by the vertical path that goes to the y-coordinate of s, followed by the path of vias that goes to s. Some of these subpaths can be empty. The second one is constructed analogously, swapping the roles of r and s. By Lemma 1, one of these $2l^2$ paths must be optimal. □

We now show how to improve on this, obtaining a linear dependence on the number of layers:

Theorem 3 *Let $c : E \to \mathbb{R}_{>0}$ depend only on direction and layer. Then, without preprocessing, one can compute $\mathrm{dist}_{(G,c)}(s, T)$ for any given $s \in V$ and given $T \subseteq V$ consisting of t rectangles in $O(tl)$ time.*

Proof Again we enumerate over all t rectangles that T is composed of, and for each such rectangle R, we determine the vertex $r \in R$ that is closest to s (geometrically) in constant time. First compute the total cost c_{z_1,z_2} of a path of vias between layer z_1 and layer z_2 for all $z_1, z_2 \in \{1, \ldots, l\}$ with $\{z_1, z_2\} \cap \{z_r, z_s\} \neq \emptyset$, where z_r and z_s denote the layers of r and s, respectively. This can easily be done in $O(l)$ time.

Now we compute the minimum cost of a path from r to s that (when traversed from r to s) consists of a path of vias, then a horizontal path, then a path of vias, then a vertical path, then a path of vias. We will then do the same with exchanging the roles of r and s, and the smaller of the two is the distance between r and s by Lemma 1.

For each layer $z \in \{1, \ldots, l\}$, consider the vertex v_z on layer z whose y-coordinate is the one of r and whose x-coordinate is the one of s. We first compute the distance $\bar{d}_z$ between r and v_z in the subgraph of G that contains no horizontal edges on the layers $1, \ldots, z-1$. By Lemma 1, a shortest path from r to v_z consists of a path of vias, a single horizontal path, and another path of vias. Hence $\bar{d}_z$ is either the sum of $c_{z_r,z}$ (which we have precomputed) and $c_z^{\leftrightarrow}$ times the difference of the x-coordinates of r and s, or $\bar{d}_{z+1} + c_{z,z+1}$ (if $z < l$), whichever is smaller. This implies that $\bar{d}_1, \ldots, \bar{d}_l$ can be computed in reverse order in total time $O(l)$.

Now we compute the distance d_z from r to v_z in G for all $z \in \{1, \ldots, l\}$. We have $d_1 = \bar{d}_1$ by definition. For $z \in \{2, \ldots, l\}$, we have $d_z = \min\{\bar{d}_z, d_{z-1} + c_{z-1,z}\}$, which allows to compute $d_1, \ldots, d_l$ in total time $O(l)$. See Fig. 4 for an illustration.

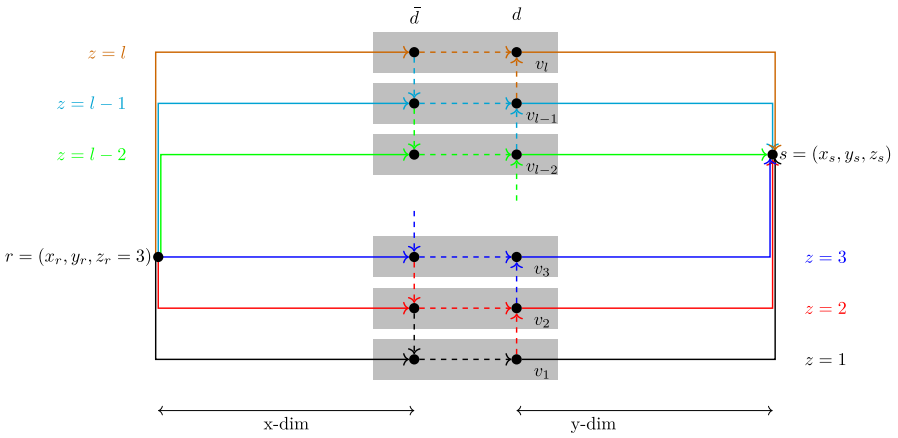

Fig. 4 Illustration of the algorithm described in the proof of Theorem 3. The vertices correspond to values computed during the algorithm, the edges correspond to the paths used to derive them. The two vertices in the same gray box are at the same geometric location $v_z = (x_s, y_r, z)$. The algorithm starts by computing $\bar{d}_l$ and then derives $\bar{d}_z$ for $z \in \{1, \ldots, l-1\}$ as the minimum of $\bar{d}_{z+1} + c_{z,z+1}$ and the cost of the path using layer z for the horizontal edges. Then it propagates the values from bottom to top to compute $d_1, \ldots, d_l$. For each $z \in \{1, \ldots, l\}$ we combine the resulting r-v_z-path with the paths indicated on the right to get an r-s-path. Finally, we take the cheapest of the l paths to obtain a shortest r-s-path

Next we compute the cost of a path from r to s that consists of a shortest path from r to v_z, a vertical path on layer z, and a path of vias. It is given by $d_z + c_{z,z_s}$ plus $c_z^{\updownarrow}$ times the difference of the y-coordinates of r and s, and thus can now be computed in constant time. Taking the minimum over all layers z yields a shortest path from r to s by Lemma 1 and thus completes the proof. □

3 Logarithmic query time in the simple model

We will now show how to achieve $O(\log(t + l))$ query time with polynomial preprocessing time but did not care about the degree of the polynomial. We will show how to do the same with only $O(t^2 l^3 \log l)$ preprocessing time. We will need two ingredients before we can prove this. The first ingredient is an efficient algorithm for computing the intersection of three-dimensional half-spaces:

Theorem 4 ([17]) *The intersection of a set of n half-spaces in three-dimensional space can be computed in $O(n \log n)$ time. If the intersection is nonempty, it is a convex polyhedron that is presented as a minimal set of inequalities, the cycle of edges surrounding each face, and the coordinates of their endpoints.*

The second ingredient is an algorithm for solving the planar point location problem, a well-studied problem in computational geometry. In the following, we regard any connected, closed part of a line in $\mathbb{R}^2$ as a *line segment* and any region whose boundary consists of a finite number of line segments as a *polygon*.

Theorem 5 ([14]) *Let L be a finite set of line segments that intersect only at their endpoints and let $(P_i)_{i\in I}$ be the (open polygonal) connected components of $\mathbb{R}^2 \setminus \bigcup L$. Then there is a data structure that requires $O(|L|\log|L|)$ preprocessing time and, given any query point $p \in \mathbb{R}^2$, can then determine an index $i \in I$ such that p lies in the closure of P_i in $O(\log|L|)$ query time.*

Point location algorithms that attain the same theoretic guarantees as [14], but successively improve practical performance and ease of implementation have been described in [6, 11, 19].

Theorems 4 and 5 can be combined in order to obtain a data structure for storing affine functions efficiently such that their pointwise minimum can be queried in logarithmic time:

Lemma 6 *Let F be a set of affine functions $f : \mathbb{R}^2 \to \mathbb{R}$ and $R{:=}[x^-, x^+] \times [y^-, y^+]$ a closed rectangle. Then there is a data structure that requires $O(|F|\log|F|)$ preprocessing time and, given any query point $p \in R$, can then determine the value $\min_{f\in F} f(p)$ in $O(\log|F|)$ query time.*

Proof We intersect the $|F|$ half-spaces $\{(x, y, \varphi) \in \mathbb{R}^3 \mid \varphi \le f(x, y)\}$ for $f \in F$ using Theorem 4. After projecting the result into the plane, we obtain a subdivision of the rectangle R into at most $|F|$ convex polygons and a minimizing function $f \in F$ for each polygon.

Second, we initialize a data structure to solve the point location problem within that subdivision.

By Euler's formula, since each vertex (with possible exception of the four corners of R) in this subdivision is incident to at least three edges, the subdivision contains at most $3|F| + 1$ line segments. Hence, this preprocessing can be implemented to run in $O(|F|\log|F|)$ time by Theorems 4 and 5.

When given a query location $p \in R$, we look up a polygon containing the query location in $O(\log|F|)$ time by Theorem 5, and evaluate the function attaining the minimum on that polygon in constant time. If the query point is on the boundary of multiple polygons, it suffices to evaluate the minimizing function of any one of these polygons. □

We can now prove the main result of this section:

Theorem 7 *Let $c : E \to \mathbb{R}_{>0}$ depend only on direction and layer, and let $T \subseteq V$ consist of t rectangles. Then there is a data structure that requires $O(t^2 l^3 \log l)$ preprocessing time and, for any given $s \in V$, can then determine $\mathrm{dist}_{(G,c)}(s, T)$ in $O(\log(t + l))$ query time.*

Proof We first interpret our instance as an instance of the general model by choosing $p = q = 0$, and then refine the grid with respect to the targets, which yields $O(t^2 l)$ tiles V_z^{ij}. We compute an independent data structure for each tile.

By Lemma 1, a shortest path from any $s \in V$ to any $r \in T$ contains at most one sequence of horizontal and at most one sequence of vertical edges. For a fixed set of directions and layers of these sequences, for example north on layer z^{N} and west on layer z^{W}, there are at most two such paths, depending on whether we first go north and

then west or vice versa, having possibly different via costs. Given a fixed tile V_z^{ij}, for every $s = (x_s, y_s, z) \in V_z^{ij}$, we are interested in the cost of a shortest such path to T, which is given by an affine function

$$f^{(i,j,z,\mathrm{NW},z^{\mathrm{N}},z^{\mathrm{W}})}(x_s, y_s) = c^{(i,j,z,\mathrm{NW},z^{\mathrm{N}},z^{\mathrm{W}})} - y_s c_{z^{\mathrm{N}}}^{\updownarrow} + x_s c_{z^{\mathrm{W}}}^{\leftrightarrow}$$

for some constant $c^{(i,j,z,\mathrm{NW},z^{\mathrm{N}},z^{\mathrm{W}})}$. To obtain the constant, one takes the minimum over all $r = (x_r, y_r, z_r) \in T$ that lie northwest of V_z^{ij}, considering the sum of $y_r c_{z^{\mathrm{N}}}^{\updownarrow} - x_r c_{z^{\mathrm{W}}}^{\leftrightarrow}$ and the via cost in the cheaper of the two cases regarding the order of the two directions. If there is no $r \in T$ northwest of V_z^{ij}, no path of the required structure exists, and the constant can be considered to be ∞.

Similarly, an affine function for each of the $1 + 4l + 4l^2$ combinations

$$(-),\ (\mathrm{E}, z^{\mathrm{E}}),\ (\mathrm{N}, z^{\mathrm{N}}),\ (\mathrm{W}, z^{\mathrm{W}}),\ (\mathrm{S}, z^{\mathrm{S}}),\ (\mathrm{NE}, z^{\mathrm{N}}, z^{\mathrm{E}}),\ (\mathrm{NW}, z^{\mathrm{N}}, z^{\mathrm{W}}),\ (\mathrm{SW}, z^{\mathrm{S}}, z^{\mathrm{W}}),\ (\mathrm{SE}, z^{\mathrm{S}}, z^{\mathrm{E}})$$

can be defined, and $f^{(i,j,z)} : (x_s, y_s) \mapsto \mathrm{dist}(s, T)$ is the pointwise minimum of these. Here $(-)$ means that we go to the target only by vias (or we are already at a target).

We next show how to compute all $O(t^2 l^3)$ affine functions in total time $O(t^2 l^3)$. We describe this for one combination $(\mathrm{NW}, z^{\mathrm{N}}, z^{\mathrm{W}})$; it works analogously for the other combinations. Here we need to compute the constants $c_z^{ij} := c^{(i,j,z,\mathrm{NW},z^{\mathrm{N}},z^{\mathrm{W}})}$ for all tiles V_z^{ij}. For each layer z, we do this from northwest to southeast. We start with $c_z^{ij} = \infty$ whenever $i = 0$ or $j = q$ and then set

$$\begin{aligned} c_z^{ij} \;= \min \Big\{ & c_z^{i,j+1},\ c_z^{i-1,j},\ \min \Big\{ \upsilon^{j+1} c_{z^{\mathrm{N}}}^{\updownarrow} - \xi^i c_{z^{\mathrm{W}}}^{\leftrightarrow} \\ & + \min \left\{ c_{z,z^{\mathrm{N}},z^{\mathrm{W}},z'},\ c_{z,z^{\mathrm{W}},z^{\mathrm{N}},z'} \right\} \mid (\xi^i, \upsilon^{j+1}, z') \in T \Big\} \Big\} \end{aligned}$$

in increasing order of $i - j$, where c_{z_1,z_2,z_3,z_4} is the total cost of the vias to go from layer z_1 to layer z_2 to layer z_3 to layer z_4; see Fig. 5 for an illustration. Since each of the t target rectangles shows up $O(l^3)$ times (once for each combination and each z), all these $O(t^2 l^3)$ affine functions can be computed in $O(t^2 l^3)$ time.

Rather than storing just a list of these $1 + 4l + 4l^2$ affine functions for each V_z^{ij}, we now build a data structure for each tile, using Lemma 6, in order to obtain a logarithmic query time. The total preprocessing time required to build these data structures is $O(t^2 l^3 \log l)$.

When given a query location $s \in V$, we find the correct tile V_z^{ij} and hence the correct data structure in $O(\log t)$ time by performing two binary searches. Then we look up the distance from s to T in $O(\log l)$ time by Lemma 6. If the query point is on the boundary of multiple tiles, it suffices to evaluate the minimizing function of any one of these tiles. □

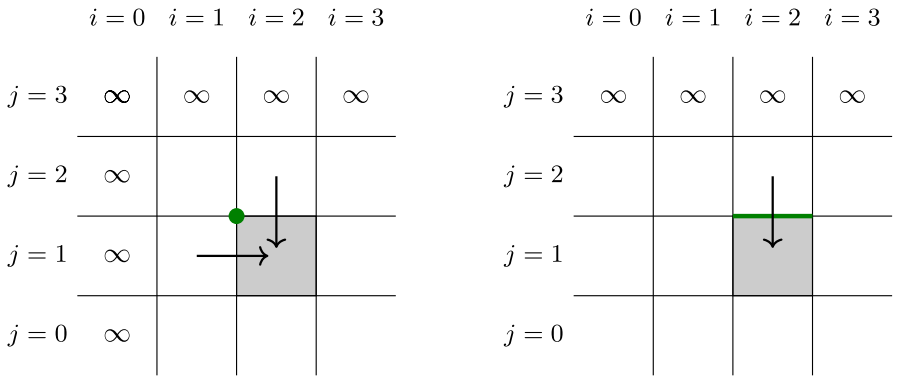

Fig. 5 Computation of the constants c_z^{ij} for a fixed layer z in the proof of Theorem 7. The left picture illustrates the propagation for the directions NW, the right picture for N. The constants corresponding to tiles that have no neighboring tiles in northwest or north direction, respectively, are set to ∞. When determining the constant c_z^{21}, the constants c_z^{22} and c_z^{11} in the NW case, and only c_z^{22} in the N case, are taken into account. Moreover, if the marked point in the NW case or the marked line segment in the N case belongs to a target rectangle in any layer z', the cost of a path to this point or line contributes to the computation of c_z^{21} as well

4 The general model

In this section, we develop an algorithm to compute the potential $\pi(v) = \mathrm{dist}_{(G,c)}(v, T)$ for any v in the general model efficiently after preprocessing. We will assume T to be consistent with the grid, i.e., we have already refined the grid if it was not. Our preprocessing will work on the horizontal and vertical line segments of the grid, i.e., the sets $\mathrm{Hor}_z^{ij} := \left\{(x, y, z) \in V_z^{ij} \mid y = v^j\right\}$ and $\mathrm{Ver}_z^{ij} := \left\{(x, y, z) \in V_z^{ij} \mid x = \xi^i\right\}$. The exposition will focus on the horizontal line segments; vertical segments can be handled analogously. Our algorithm consists of two preprocessing steps and a query step. The first preprocessing step is a variant of Dijkstra's algorithm. For its correctness, the following observation about the structure of shortest paths, similar to Lemma 1 for the simple model, is essential:

Lemma 8 *Let $c : E \to \mathbb{R}_{>0}$ depend on tile and direction, let $T \subseteq V$ be consistent with the grid, and $s \in V \setminus T$. Then there is a shortest path P from s to T in (G, c) that uses only one type of edges (either horizontal, vertical or via) before entering some tile in which s does not lie.*

Proof By the same argument as in Lemma 1, there is a shortest path P such that every subpath of P that is entirely in the interior of one column $\{(x, y, z) \in V \mid \xi^i \le x \le \xi^{i+1}\}$ contains at most one horizontal sequence of edges and every subpath of P that is entirely in the interior of one row $\{(x, y, z) \in V \mid v^j \le y \le v^{j+1}\}$ contains at most one vertical sequence of edges. It is easy to see that such a path satisfies the claim. □

Our algorithm will first compute $\mathrm{dist}_{(G,c)}(s, T)$ for all s lying in horizontal segments of the grid. By applying Lemma 8 inductively, we may assume that the corresponding shortest paths never use a horizontal edge or a via in the interior of a tile.

In the variant of Dijkstra's algorithm that we apply, we do not mark all vertices whose labels are guaranteed to be permanent explicitly, but the set of these vertices at some point in the algorithm is implicitly given: it consists of all vertices that have a smaller label than the one considered in the current iteration. In every iteration, several vertices might be added to this set and several updates of neighboring vertices are performed. This will be useful in order to decrease the required number of iterations significantly compared to the regular Dijkstra's algorithm, while the additional updates in every iteration can be performed with negligible overhead. This is possible because all horizontal edges along some horizontal segment have the same cost, and hence, the corresponding labels can be represented by affine functions that can be stored and updated very efficiently. Instead of vertices we will store affine functions in a heap (priority queue), and the *key* of such a function will be the minimum relevant function value.

More precisely, for each horizontal segment Hor_z^{ij}, the algorithm maintains a set F_z^{ij} of affine functions $f\colon [\xi^i, \xi^{i+1}] \to \mathbb{R}_{\geq 0}$ such that each value $f(x)$ corresponds to the cost of a path between (x, υ^j, z) and T. At any point during the algorithm, for every vertex (x, υ^j, z), the minimum value $\min\{f(x) \mid f \in F_z^{ij}\}$ can be considered to be its current label. If $\xi^i = \xi^{i+1}$, we simply store that value. Otherwise we keep only those functions that are not dominated, i.e., attain the pointwise minimum in more than one point. The key of a non-dominated function $f \in F_z^{ij}$ is the minimum function value in the interval in which f attains the pointwise minimum. For storing the sets F_z^{ij}, we will use the following result. See Fig. 6 for an illustration.

Lemma 9 *Let $x^-, x^+ \in \mathbb{R}$ be given with $x^- < x^+$. We can maintain a data structure that*

- *stores a set F of affine functions $g : \mathbb{R} \to \mathbb{R}$ such that the set $\{x \in [x^-, x^+] : g(x) = \min_{f \in F} f(x)\}$ of all points where the function attains the minimum is an interval $[x_g^-, x_g^+]$ with $x_g^- < x_g^+$,*
- *stores for every function g the interval $[x_g^-, x_g^+]$ and the minimum value $\mathrm{key}(g) := \min\{g(x_g^-), g(x_g^+)\}$ of g on this interval, and*
- *can be updated in $O((D+1)\log(D+|F|))$ time when adding a new function, where D is the number of functions that have to be removed from F because they are dominated.*

Moreover, each update increases the key of at most one function that remains in F, and never decreases a key.

Proof We store the set of affine functions as a binary search tree in which they are sorted by their slopes. When a function g is added, we check whether g is dominated and otherwise insert it into the search tree in time $O(\log |F|)$. Then, starting from g, we iterate forward and backward, deleting all functions f that are identical to g or dominated by g on the whole interval $[x_f^-, x_f^+]$. Each of these operations requires time $O(\log |F|)$. If a function f is partly dominated by g, then its interval and, if necessary, its key are updated. It can be required to shorten two intervals, left and right of the interval of g. However, only one key can increase due to the concavity of the pointwise minimum. □

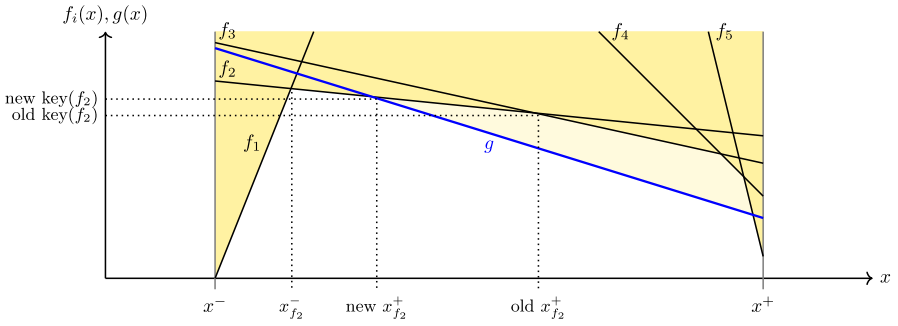

Fig. 6 Update of the data structure described in Lemma 9 when adding an affine function g to a set $F = \{f_1, f_2, f_3, f_4, f_5\}$: here f_3 and f_4 are deleted, $x^+_{f_2}$ decreases, key(f_2) increases, and $x^-_{f_5}$ increases

In addition to storing each set F_z^{ij} as specified in Lemma 9, we maintain a binary heap representing all functions in $\bigcup\{F_z^{ij} \mid i \in \{0, \dots, p\}, j \in \{1, \dots, q\}, z \in \{1, \dots, l\}\}$ that have not been processed yet, using the keys defined in Lemma 9. The functions that are added to or removed from some F_z^{ij} must be added to or removed from the heap at the same time, and whenever a key changes, it must be updated also in the heap.

The algorithm starts by initializing $F_z^{ij} := \emptyset$ for all $i \in \{0, \dots, p\}$, $j \in \{1, \dots, q\}$, and $z \in \{1, \dots, l\}$. If $\text{Hor}_z^{ij} \subseteq T$, we add the constant function $x \mapsto 0$ to the corresponding set F_z^{ij}. If not the whole segment, but one (or both) of its endpoints lies in T, we add the affine function describing the distance to this endpoint, i.e., $x \mapsto \min\{c_z^{ij\leftrightarrow}, c_z^{i(j-1)\leftrightarrow}\} \cdot (x - \xi^i)$ or $x \mapsto \min\{c_z^{ij\leftrightarrow}, c_z^{i(j-1)\leftrightarrow}\} \cdot (\xi^{i+1} - x)$.

In every iteration, a function f with minimum value key(f) is chosen and removed from the heap. The function f describes the labels of a subset of some horizontal segment Hor_z^{ij}, corresponding to the interval $[x_f^-, x_f^+]$. We now propagate the labels from these vertices to the neighboring horizontal segments by computing at most six new affine functions:

(down) If $z > 1$, add the function $x \mapsto f(x) + \min\{c_{z-1,z}^{ij}, c_{z-1,z}^{i(j-1)}\}$ to F_{z-1}^{ij}.
(up) If $z < l$, add the function $x \mapsto f(x) + \min\{c_{z,z+1}^{ij}, c_{z,z+1}^{i(j-1)}\}$ to F_{z+1}^{ij}.
(south) If $j > 1$, add the function $x \mapsto f(x) + c_z^{i(j-1)\updownarrow} \cdot (\upsilon^j - \upsilon^{j-1})$ to $F_z^{i(j-1)}$.
(north) If $j < q$, add the function $x \mapsto f(x) + c_z^{ij\updownarrow} \cdot (\upsilon^{j+1} - \upsilon^j)$ to $F_z^{i(j+1)}$.
(west) If $i > 0$, add the function $x \mapsto f(\xi^i) + \min\{c_z^{(i-1)j\leftrightarrow}, c_z^{(i-1)(j-1)\leftrightarrow}\} \cdot (\xi^i - x)$ to $F_z^{(i-1)j}$.
(east) If $i < p$, add the function $x \mapsto f(\xi^{i+1}) + \min\{c_z^{(i+1)j\leftrightarrow}, c_z^{(i+1)(j-1)\leftrightarrow}\} \cdot (x - \xi^{i+1})$ to $F_z^{(i+1)j}$.

The algorithm stops when the heap is empty. For an example run of the algorithm, see Figure 7. Its correctness, i.e., the fact that, after termination, for all $i \in \{0, \dots, p\}$,

$j \in \{1, \ldots, q\}$, $z \in \{1, \ldots, l\}$, and $(x, \upsilon^j, z) \in \mathrm{Hor}_z^{ij}$, we have $\min\{f(x) \mid f \in F_z^{ij}\} = \mathrm{dist}_{(G,c)}((x, \upsilon^j, z), T)$, follows from the following two lemmas.

Lemma 10 *Let $i \in \{0, \ldots, p\}$, $j \in \{1, \ldots, q\}$, $z \in \{1, \ldots, l\}$, and $(x, \upsilon^j, z) \in \mathrm{Hor}_z^{ij}$. If, at any point during the algorithm, an affine function f is added to F_z^{ij}, then there is an (x, υ^j, z)-T-path of cost at most $f(x)$. In particular, $f(x) \geq \mathrm{dist}_{(G,c)}((x, \upsilon^j, z), T)$ holds for all $f \in F_z^{ij}$.*

Proof We prove the assertion by induction on the order in which the affine functions are added (for all combinations of i, j, z, and x at once). If f is added during the initialization phase, then there is a path with the desired property that only consists of horizontal edges. Otherwise, f is added during some iteration later on and derived from some $g \in F_{z'}^{i'j'}$. Then it suffices to build (possibly zero) edges in one of the six possible directions until a vertex $(x', \upsilon^{j'}, z') \in \mathrm{Hor}_{z'}^{i'j'}$ is reached. By the induction hypothesis, there is an $(x', \upsilon^{j'}, z')$-T-path of cost at most $g(x')$. The propagation ensures that $f(x)$ is at least the cost of this path combined with the straight series of edges. □

Lemma 11 *Let $K \in \mathbb{R}_{\geq 0}$ be the key of a function that is chosen in some iteration of the algorithm (or $K = \infty$ if the algorithm has terminated) and let $i \in \{0, \ldots, p\}$, $j \in \{1, \ldots, q\}$, $z \in \{1, \ldots, l\}$, and $x \in \{\xi^i, \ldots, \xi^{i+1}\}$ such that $\mathrm{dist}_{(G,c)}((x, \upsilon^j, z), T) < K$. Then there is a function $f \in F_z^{ij}$ with $f(x) = \mathrm{dist}_{(G,c)}((x, \upsilon^j, z), T)$.*

Proof Let $s = (x, \upsilon^j, z)$. We prove the result by induction on $\mathrm{dist}_{(G,c)}(s, T)$. Note that there can be different i and j with $s \in \mathrm{Hor}_z^{ij}$, both because x can be a grid coordinate and because there can be two grid coordinates at υ^j. Our inductive step consists of two parts. In the first part, we will disregard the given i and j and instead show it is possible to choose i and j such that the inductive hypothesis holds for the given point s.

If $s \in T$, then an affine function as desired was added to some F_z^{ij} in the initialization phase. By Lemma 10, the minimum value of the functions in F_z^{ij} at x cannot decrease. Because we only remove dominated functions, there will always be a function in F_z^{ij} attaining this value.

If $s \notin T$, then consider a shortest s-T-path P. By Lemma 8, we may assume that P starts with a straight series of edges to the first point that lies in a tile not containing s. Denote this point by s'. Consider i', j', and z' with $s' \in \mathrm{Hor}_{z'}^{i'j'}$. By the induction hypothesis, a function g with $g(x') = \mathrm{dist}_{(G,c)}(s', T)$ is contained in $F_{z'}^{i'j'}$. Using Lemma 10, we derive that g is not strictly dominated in x'. Hence, $\mathrm{key}(g) \leq g(x') = \mathrm{dist}_{(G,c)}(s', T) < \mathrm{dist}_{(G,c)}(s, T) < K$. Since keys never decrease during the algorithm by Lemma 9, g was chosen and removed from the heap in a prior iteration. Now note that it is possible to choose i, j, i', and j' such that $s \in \mathrm{Hor}_z^{ij}$ and the propagation of g adds a function f satisfying $f(x) = c(P)$ to F_z^{ij}. This concludes the first part of the inductive step.

For the second part of the inductive step, consider all i and j with $s \in \mathrm{Hor}_z^{ij}$. Note that the possible choices of i and j are independent and consecutive. We know from

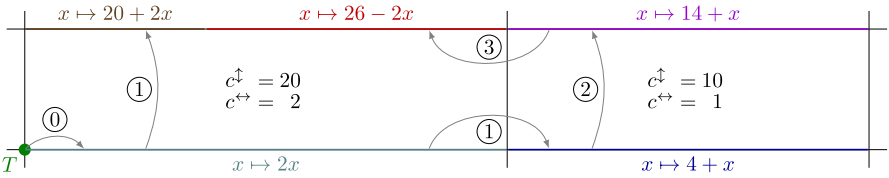

Fig. 7 Example run of the algorithm computing the distance from all horizontal line segments to T. The instance consists of two horizontally adjacent tiles (i.e., $p = 3$, $q = 2$, and $l = 1$). The coordinates of these tiles are $\xi^1 = 0$, $\xi^2 = 4$, $\xi^3 = 7$, $\upsilon^1 = 0$, and $\upsilon^2 = 1$. The target $T = \{(0, 0, 1)\}$ consists of the single point in the bottom left corner of the left tile. We disregard the outside tiles (by setting their costs to infinity). All other costs are as written in the centers of the respective tiles. During the algorithm, five affine functions are added to the four horizontal segments. The horizontal segments are colored by the function attaining the minimum in the end of the algorithm. The incoming arrow depicts the propagation by which that function was added and is numbered by the iteration of the algorithm (where 0 stands for initialization) (color figure online)

the first part that there are some i_1 and j_1 such that there is a function $f_1 \in F_z^{i_1 j_1}$ with $f_1(x) = \mathrm{dist}_{(G,c)}(s, T) < K$. Since this function is not dominated in x, we conclude $\mathrm{key}(f_1) < K$. This means that this function has already been propagated and has added functions with the same value at x to all adjacent F_z^{ij} for which $s \in \mathrm{Hor}_z^{ij}$. Successive application concludes the proof. □

Note that both steps are necessary in the above proof. Figure 7 contains one such situation: we are looking for a function $h \in F_1^{12}$ that gives us the correct distance from the top right corner of the left tile to T, i.e., with $h(4) = \mathrm{dist}_{(G,c)}((4, 1, 1), T) = 18$. Since the vertical cost is lower in the right tile, the first step gives us the function $g : x \mapsto 4 + x \in F_1^{21}$, which is propagated to $f : x \mapsto 14 + x \in F_1^{22}$, which in turn is propagated to the desired function $h : x \mapsto 26 - 2x \in F_1^{12}$.

To ensure that the algorithm terminates and has the desired running time, we first show:

Lemma 12 *Let $k' := \min\{k, (q + 1)l\}$. The number of slopes of functions ever added to F_z^{ij} is at most $2k' + 1$. In particular, this also bounds the cardinality of F_z^{ij} at any stage.*

Proof By induction, the slope of every affine function added to F_z^{ij} during the algorithm is either zero or an element of $\{+c_{z'}^{ij'\leftrightarrow}, -c_{z'}^{ij'\leftrightarrow} \mid j' \in \{0, \ldots, q\}, z' \in \{1, \ldots, l\}\}$. Hence, if there were more than $2k' + 1$ functions in F_z^{ij}, then there would be two different functions having the same slope. But then one of them is strictly dominated by the other one and would be removed from F_z^{ij}, contradicting the specification in Lemma 9. □

In order to achieve the desired running time, we bound the number of iterations by $(p + 1)q(2k' + 1)l$. Each iteration chooses and removes a function f from the heap. Let $(x, \upsilon^j, z) \in \mathrm{Hor}_z^{ij}$ such that $f(x) = \mathrm{key}(f)$. By Lemma 11, $f(x) = \mathrm{dist}((x, \upsilon^j, z), T)$ and, by Lemma 10, each function g added to F_z^{ij} later on satisfies $g(x) \geq \mathrm{dist}_{G,c}((x, \upsilon^j, z), T)$. This means no other function with the same slope as f can ever be an element of F_z^{ij} in the future. Since the number of slopes of functions

in $F_{z'}^{i'j'}$ was bounded by $2k'+1$ in Lemma 12, we obtain the claimed bound on the number of iterations.

Finally, we need to implement each iteration in amortized time $O(\log(p+q+l))$. Since each iteration generates at most six new functions and the number of iterations is bounded by $(p+1)q(2k'+1)l$, at most $2(p+1)ql+6(p+1)q(2k'+1)l$ functions are added in total, including the initialization. This also bounds the size of the binary heap to $O(pq^2l^2)$ such that each heap operation can be performed in time $O(\log(p+q+l))$. By Lemma 9, each added function causes at most one increase-key operation. The total number of deletions is clearly bounded by the total number of insertions. Thus, the total time needed for the heap operations is given by $O(pqk'l\log(p+q+l))$. The same holds for the time required for the updates of the sets F_z^{ij} by Lemma 9. This proves:

Theorem 13 *There is an algorithm that computes for each horizontal segment* Hor_z^{ij} *a set* F_z^{ij} *of at most* $2k'+1$ *affine functions such that* $\min\left\{f(x) \mid f \in F_z^{ij}\right\} = \mathrm{dist}_{(G,c)}((x,v^j,z),T)$ *for all* $(x,v^j,z) \in \mathrm{Hor}_z^{ij}$. *The algorithm can be implemented to run in* $O(pqk'l\log(p+q+l))$ *time, where again* $k' = \min\{k,(q+1)l\}$. □

We will now consider how to implement queries. We do so for all pairs of i and j independently. If we wanted to execute a query right after executing the algorithm described by Theorem 13 without any further preprocessing, the best we could do is $O(\log(p+q)+l\log k)$ query time: we first compute i and j by binary search. In each of the $4l$ segments $\mathrm{Hor}_{z'}^{ij}$, $\mathrm{Ver}_{z'}^{(i+1)j}$, $\mathrm{Hor}_{z'}^{i(j+1)}$, and $\mathrm{Ver}_{z'}^{ij}$ ($z' \in \{1,\dots,l\}$), we then compute the closest point r to the query location s. The distance from s to r can be computed in amortized constant time and the distance from r to T can be looked up in $O(\log k)$ time in the data structure storing the affine functions of that segment.

The above can be seen as an evaluation of the minimum of $O(\min\{k,(p+q+1)l\}l)$ affine functions. Hence it might be worthwhile building up the data structure described by Lemma 6 in an additional preprocessing step. This would speed up our queries to $O(\log(p+q+l))$ time, however at the cost of an additional $O(\min\{k,(p+q+1)l\}l\log(p+q+l))$ preprocessing time per tile.

The following result will obtain a trade-off between these two alternatives. By choosing the trade-off factor $0 < \epsilon \le 1$ to be a small constant, we obtain a query time of $O(\log(p+q+l))$ after a preprocessing time which is arbitrarly close to $O(pq\min\{k,(p+q+1)l\}l\log(p+q+l))$.

Theorem 14 *Let* $0 < \epsilon \le 1$, *let* $c : E \to \mathbb{R}_{>0}$ *depend on tile and direction, and let* $T \subseteq V$ *be consistent with the grid. Then there is a data structure that requires* $O(pq\min\{k,(p+q+1)l\}l^{1+\epsilon}\frac{1}{\epsilon}\log(p+q+l))$ *preprocessing time and, for any given* $s \in V$, *can then determine* $\mathrm{dist}_{(G,c)}(s,T)$ *in* $O(\log(p+q)+\frac{1}{\epsilon}\log(k+l))$ *query time.*

Proof We first apply Theorem 13 to compute $\mathrm{dist}_{(G,c)}(v,T)$ for all v in some Hor_z^{ij} and (analogously) for all v in some Ver_z^{ij}. Now our preprocessing will consider all combinations of $i \in \{0,\dots,p\}$ and $j \in \{0,\dots,q\}$ separately. Given a query location

$(x, y, z) \in V_z^{ij}$, i and j can be determined in $O(\log(p+q))$ time by binary search. Hence we fix i and j from now on.

We refer to the union of the up to $4l$ segments $\mathrm{Hor}_{z'}^{ij}$, $\mathrm{Ver}_{z'}^{(i+1)j}$, $\mathrm{Hor}_{z'}^{i(j+1)}$, and $\mathrm{Ver}_{z'}^{ij}$ ($z' \in \{1, \ldots, l\}$) as the *boundary*. First suppose that a shortest s-T-path does not touch the boundary. Since T is consistent with the grid, such a path consists of vias only. The cost of such vias-only paths can be easily precomputed in $O(pql)$ total preprocessing time, allowing for $O(1)$ query time.

Now suppose that a shortest s-T-path touches the boundary at least once. Consider a (without loss of generality) a horizontal segment H of the boundary that is touched first by one such path. By Lemma 8, we may pick our shortest path such that it starts with a sequence of vias to the layer of H, followed by a sequence of vertical edges to H. Hence, we can compute the cost of the path until it first touches H as an affine function in y, where $s = (x, y, z)$. The rest of the path is a shortest path from some point in H to T, so we already computed its cost as a minimum of $O(\min\{k, (q+1)l\})$ affine functions in x, by Theorem 13. By iterating over each of these affine functions for every horizontal and vertical segment in the boundary, we can express the cost of a shortest s-T-path as a minimum of $O(\min\{k, (p+q+1)l\}l)$ affine functions in x and y. Note that the only dependence of these functions on z is the cost of the initial via stack. We will exploit this now.

Instead of building a separate data structure for each layer, each involving all these affine functions, we distinguish between the cases whether the shortest path from s to T begins without vias, with vias up to a higher level, or with vias down to a lower layer. For the first case, we build up the point location data structure just as in Lemma 6, but each involving only the $O(\min\{k, (p+q+1)l\})$ affine functions on the boundary segments on that layer. We call these data structures $D^{=z}$ for $z \in \{1, \ldots, l\}$.

For the other two cases, we build data structures $D^{\uparrow[a,b]}$ and $D^{\downarrow[a,b]}$ for some $1 \le a \le b \le l$. Here $D^{\uparrow[a,b]}$ considers query locations on all layers $z \in \{1, \ldots, a\}$ and all boundary segments in the layer range $\{a, \ldots, b\}$, i.e., paths from (x, y, z) that begin with a (possibly empty) via stack from layer z up to some layer $z' \in \{a, \ldots, b\}$ and then proceed via a straight horizontal or vertical path to the boundary. Similarly, $D^{\downarrow[a,b]}$ considers query locations on layers $z \in \{b, \ldots, l\}$ and boundary segments in the layer range $\{a, \ldots, b\}$. Note that such a data structure involves $O((b+1-a)\min\{k, (p+q+1)l\})$ affine functions, hence, by Lemma 6, it can be constructed in $O((b+1-a)\min\{k, (p+q+1)l\}\log(p+q+l))$ time and then allows for queries in $O(\log(k+l))$ time.

The main advantage is that we can use the same data structure $D^{\uparrow[a,b]}$ for all layers $z \in \{1, \ldots, a\}$ because the cost of the via stack from layer z to layer a is a constant term that depends only on z. We can design the data structures so that each affine function on the boundary shows up only in at most $2l^\epsilon$ (instead of l) of these data structures.

To this end, we consider a balanced arborescence A whose leaves are the layers $1, \ldots, l$, such that A has maximum out-degree $d = \lceil l^\epsilon \rceil$ and depth $\lceil \frac{1}{\epsilon} \rceil$, and, for every vertex v of A, the set of leaves reachable from v in A is a consecutive range $L(v)$. Let $\mathrm{parent}(v)$ denote the parent of v (unless v is the root). For every vertex v except for the root, let $L^{>}(v) := \{z \in L(\mathrm{parent}(v)) \mid z > z' \text{ for all } z' \in L(v)\}$ and

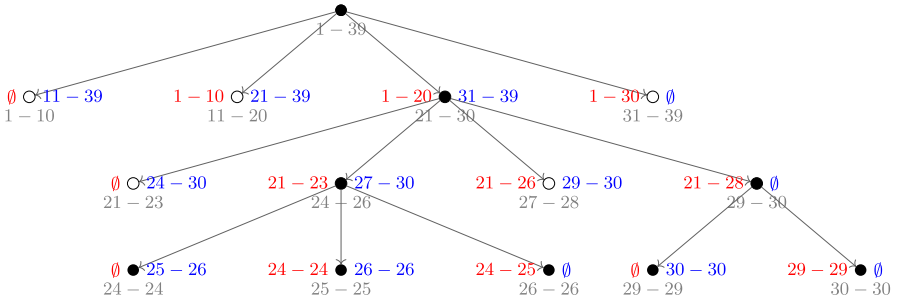

Fig. 8 A possible choice for the aborescence A in the proof of Theorem 14 if $l = 39$ and $\epsilon = \frac{1}{3}$. For the sake of clarity, the branches below the five white vertices were omitted. Below each node v, there is a gray label describing the range $L(v)$. If v is not the root, there are also a red label left of v describing the layer range $L^<(v)$ and a blue label right of v describing the layer range $L^>(v)$ (color figure online)

$L^<(v) := \{z \in L(\text{parent}(v)) \mid z < z' \text{ for all } z' \in L(v)\}$. Then we store $D^{\uparrow L^>(v)}$ and $D^{\downarrow L^<(v)}$, unless this layer range is empty; see Fig. 8.

To answer a query for a point on layer z, we ask $D^{=z}$ and then traverse the path from z to the root in the arborescence A, and for each vertex v on that path (except for the root), we ask $D^{\uparrow L^>(v)}$ and $D^{\downarrow L^<(v)}$. We have to query at most $2\lceil \frac{1}{\epsilon} \rceil + 1$ data structures, and each of these queries takes $O(\log(k + l))$ time.

To bound the preprocessing time, we see that each layer z appears in the layer range of only d data structures on each level of the arborescence (one per vertex whose parent's layer range contains z), and hence $O(l^\epsilon \frac{1}{\epsilon})$ overall. Hence, the total preprocessing time is $O(\min\{k, (p+q+1)l\} l^{1+\epsilon} \frac{1}{\epsilon} \log(p+q+l))$. Since we do this for all i and j, the theorem follows. □

Corollary 15 *Let $0 < \epsilon \leq 1$, let $c : E \to \mathbb{R}_{>0}$ depend on tile and direction, and let $T \subseteq V$, not necessarily consistent with the grid. Then there is a data structure that requires $O((p+t)(q+t)\min\{k, (p+q+1)l\} l^{1+\epsilon} \frac{1}{\epsilon} \log(p+q+l+t))$ preprocessing time and, for any given $s \in V$, can then determine $\text{dist}_{(G,c)}(s, T)$ in $O(\log(p+q+t) + \frac{1}{\epsilon} \log(k+l))$ query time.*

Proof Refine the grid with respect to the targets, then apply Theorem 14 to ϵ and the refined instance. Note that this refinement does not increase the number of different costs. □

We remark that the same ideas used in this section could be applied in the simple model to obtain a faster preprocessing time of $O(t^2 l^{2+\epsilon} \frac{1}{\epsilon} \log l)$ at the cost of a slower query time of $O(\log t + \frac{1}{\epsilon} \log l)$.

5 Practical aspects

5.1 Implementation

With some modifications that we will describe below, we implemented the algorithms presented in the previous sections as part of BonnRoute [1, 2, 7, 12], a detailed router

developed at the University of Bonn in joint work with IBM. BonnRoute is the main detailed routing tool used by IBM for the design of its processor chips.

Up to parallelization and conflict resolution, BonnRoute routes one net after the other. Each net is routed by iteratively connecting two of its components by a path until the net is fully connected, i.e., one component remains. The path search is the algorithmic core of BonnRoute and requires approximately 80–90 % of the total runtime.

To ensure that the layout can be manufactured, certain design rules must be obeyed. For example, two vias must not be too close to each other even if they belong to the same net. Shortest paths in the detailed routing graph often correspond to wirings that violate design rules. Respecting even simple design rules is NP-hard [1]. BonnRoute uses a framework consisting of multiple components for avoiding violations. First, every computed path is handed to a post-processing routine, which attempts to resolve violations locally. Second, we apply multi-labeling, i.e., we search for shortest paths in a modified graph that can have multiple copies of each vertex (and different edges). The modifications are done in such a way that certain design rule violations are avoided. Finally, we impose restrictions to avoid violations at the start and end of a path. For further implementation details, see [1, 2].

All experiments were performed on the same AMD EPYC 7601 machine with 64 CPUs and 1024 GB main memory using 64 threads. Table 2 gives an overview of our testbed. It consists of nine real-world instances from three recent IBM processor chips in 7 nm and 5 nm technology nodes. We started all experiments on the same instance from the same snapshot, which was taken right before the detailed routing. At this point, a (three-dimensional) global routing and possibly an allowed layer range were already computed for each net.

We use edge costs as they have been developed for many years in real design practice. They have three main components. The first component is called the base cost. The base cost does not depend on the net and models the amount of routing resources consumed by a path. Wiring against the preferred direction of a layer (if allowed at all) is ten times as expensive as wiring in the preferred direction. Apart from that, wires in x- and y-direction have the same base cost on all layers. The base cost of vias is chosen such that a via bridge, i.e., a path consisting of two vias on the same layer and a single segment of wiring in preferred direction between them, is cheaper than the direct connection between the two endpoints if and only if it blocks strictly fewer additional tracks. This means that the precise via costs depend heavily on the precise design rules and track patterns of the technology. On the highest layers, the base cost of vias can be more than ten times more expensive than on the lowest layers. This is because the thicker wires on the high layers require a track pattern with larger spacing. The second component of the cost function is an additive penalty, increasing the cost of wires outside of the assigned layer range. This is a heuristic approach to avoid timing failures due to wires on lower layers having more resistance. Finally, the third component serves to restrict our path search to vertices that are inside of the area corresponding to the global routing solution. In the context of our general cost model, this can be done by setting the cost to infinity outside of this area.

Table 3 compares the performance of path searches with the original edge costs and with the reduced costs using three feasible potentials. Each of these three potentials is

Table 2 Testbed consisting of nine real-world instances

Chip	Tech	l	$\lvert V\rvert$ 10^9	Area mm^2	Wires m	Vias 10^6	Nets 10^6	Pins 10^6	Calls 10^6	t	pql
A1	7 nm	10	1.0	0.08	2.3	3.50	0.37	1.15	0.87	1.83	3018
A2	5 nm	10	1.1	0.09	3.2	3.22	0.29	0.92	0.68	1.98	3180
A3	7 nm	10	1.4	0.10	2.8	2.75	0.26	0.78	0.56	1.94	3074
B1	5 nm	16	4.7	0.36	9.6	6.67	0.63	1.79	1.38	2.24	10868
B2	7 nm	16	15.9	1.20	28.6	18.82	1.73	5.00	3.98	2.09	10410
B3	7 nm	16	36.7	2.77	24.8	14.04	1.37	3.73	2.75	2.51	12123
C1	7 nm	16	94.4	6.52	26.8	1.78	0.14	0.31	0.29	6.49	28739
C2	7 nm	18	244.7	16.73	97.2	8.19	0.57	1.21	2.60	7.46	52052
D1	7 nm	16	9615.9	601.97	178.3	10.30	0.88	1.86	1.96	5.77	19065

Tech refers to the naming of the technology nodes by the foundry. The vertex set V of the detailed routing graph is the set of all locations that are on track with respect to at least one wire type on the current layer and would be on track when projected to at least one adjacent layer. Both (total length of) Wires and (total number of) Vias refer to the detailed routing computed by BonnRoute. Calls is the number of calls to our Dijkstra implementation. Since BonnRoute may try out different side constraints to find a path with as few design rule violations as possible, this does not match the number of computed paths. t and pql are the arithmetic mean over all calls to the preprocessing

the distance to T in the same supergraph G of G', but with respect to different edge costs c. For the simple and the general model, our implementation differs from the description in the previous sections as described below.

- Since t is usually small (3.3 on average), we iterate over all target rectangles in T for every query. When computing the distance from the query location to one of the target rectangles in the **simple** model, we know the start and end layer. If we guess the lowest and highest layer used by a shortest path, the distance can be computed in constant time (Proposition 2). Due to the special structure of our cost function, at most l combinations need to be considered. In a preprocessing step, we compute those combinations that can be optimal for some query locations.
- The implementation of the **general** model uses a modified version of the algorithm used in Theorem 13. Since the distance from a single tile to T can be expressed as a minimum of very few affine functions (on every instance, the average is below 1.2), it is more efficient to compute these functions instead of the distance from the horizontal and vertical segments to T. On the other hand, instead of Lemma 9, we need to use a more complicated data structure to maintain the set of non-dominated functions. We store the convex polygon of points on which each function attains the minimum. This way, insertion can be implemented to run in $O(|F|\log|F|)$ time. During each query, we find the correct tile in $O(\log(p+q+t))$ time using binary search and evaluate all non-dominated functions of that tile on the query location to compute the minimum.

The results show that the general potential performs significantly better than the simple potential, which already performs much better than the ℓ_1-distance potential. Both the number of labels and the runtime improve on every instance, even when considering

Table 3 Performance of four different feasible potentials in our testbed

Chip	Potential	Preprocessing	All Dijkstra calls		Standard Dijkstra calls		Total BonnRoute
			Runtime	Labels	Runtime	Labels	Wall time
		h:mm	h:mm	10^9	h:mm	10^9	h:mm
A1	Without	0:00	19:12	19.7	15:21	18.2	0:35
A1	ℓ_1-distance	0:00	6:05	5.8	3:53	4.9	0:22
A1	Simple	0:00	4:28	4.0	2:32	3.2	0:21
A1	General	0:47	3:46	2.9	1:49	2.1	0:20
A2	Without	0:00	15:04	17.0	11:20	15.2	0:28
A2	ℓ_1-distance	0:00	6:16	6.5	3:41	5.2	0:21
A2	Simple	0:00	5:37	5.6	3:06	4.4	0:20
A2	General	0:33	4:34	4.0	2:09	2.8	0:19
A3	Without	0:00	15:25	16.8	12:14	15.2	0:27
A3	ℓ_1-distance	0:00	5:38	5.9	3:39	4.9	0:17
A3	Simple	0:00	5:02	5.3	3:10	4.3	0:17
A3	General	0:27	3:58	3.6	2:09	2.6	0:19
B1	Without	0:00	62:46	59.1	37:15	44.9	1:42
B1	ℓ_1-distance	0:00	37:15	31.8	16:37	20.6	1:17
B1	Simple	0:03	31:43	26.7	13:14	16.9	1:13
B1	General	3:12	26:23	20.1	9:33	11.5	1:11
B2	Without	0:00	194:21	168.5	119:21	136.2	5:13
B2	ℓ_1-distance	0:00	108:31	87.1	54:33	64.0	3:55
B2	Simple	0:10	91:56	71.7	41:35	50.2	3:40
B2	General	9:37	77:58	54.7	30:33	35.2	3:38
B3	Without	0:00	143:39	141.3	109:28	121.7	4:08
B3	ℓ_1-distance	0:00	78:39	74.7	50:49	59.3	3:15

Table 3 continued

Chip	Potential	Preprocessing h:mm	All Dijkstra calls		Standard Dijkstra calls		Total BonnRoute
			Runtime h:mm	Labels 10^9	Runtime h:mm	Labels 10^9	Wall time h:mm
B3	Simple	0:07	60:15	56.7	35:33	43.6	2:57
B3	General	6:48	45:18	38.5	22:48	26.9	2:54
C1	Without	0:00	111:55	129.4	86:24	113.0	2:20
C1	ℓ_1-distance	0:00	102:23	92.0	77:30	77.9	2:20
C1	Simple	0:00	86:21	78.9	63:16	66.1	1:56
C1	General	0:57	68:21	59.4	45:50	48.0	1:45
C2	Without	0:00	2331:30	1378.3	290:01	350.9	41:40
C2	ℓ_1-distance	0:00	2136:23	1110.2	309:56	274.6	38:36
C2	Simple	0:08	1997:04	1024.7	252:46	229.7	36:36
C2	General	16:59	1942:03	935.8	206:34	180.6	36:06
D1	Without	0:00	5039:08	3315.9	3937:44	2818.2	86:14
D1	ℓ_1-distance	0:00	2976:07	1452.6	2015:02	1106.8	53:45
D1	Simple	0:06	1909:23	958.0	1102:38	663.5	36:31
D1	General	7:16	1435:37	796.8	773:07	538.7	29:08
Sum	Without	0:00	7933:04	5246.5	4619:12	3633.9	142:52
Sum	ℓ_1-distance	0:02	5457:19	2867.0	2535:45	1618.6	104:12
Sum	Simple	0:39	4191:51	2232.0	1517:52	1082.4	83:54
Sum	General	46:40	3608:00	1916.2	1094:36	848.9	75:44

In the rows **without** potential, each query returns 0 in constant time. When using **ℓ_1-distance**, the distances in x- and y-direction are scaled by the minimal $c_z^{\leftrightarrow}$ and $c_z^{\updownarrow}$, respectively, over all z. An $O(l)$ preprocessing computes these two numbers and the total cost $c_{1,z}$ of vias from layer 1 to each layer z; for the distance of two points on layers $z < z'$, we then use $c_{1,z'} - c_{1,z}$. The query returns the ℓ_1-distance to T in $O(t)$ time by iterating over the targets. In the **simple** and **general** rows, the shortest distance to T in the respective models is returned. Here the only difference between the two is that the general model restricts to the area corresponding to the global routing solution (outside of it, the costs are infinite). Implementation details for the last two potentials are described in the text. Runtimes are summed over all 64 threads except for the last column, which shows the total wall time of the overall BonnRoute run

the additional preprocessing time. The relative improvement differs a lot between different instances. Most of this difference can be explained by some situations in which we get only minor improvements by our potentials:

- If no path is found, all reachable vertices in the graph are labeled. None of the potentials show any improvement on these instances. In fact, the path searches without potential are the fastest since they do not need any query time, with a total of 285 hours (summed over all instances and all 64 threads). With the three potentials, these path searches take a total of 329, 337, and 331 hours, respectively.
- After a path search failed, BonnRoute may perform a backup path search which allows routing through existing wires at high cost (and then would remove (*rip up*) such wires and try to re-route them). Since these rip-up costs are not modeled in any of our potentials, a large portion of the graph may be labeled regardless of which potential is used. On such instances, the order of the potentials regarding their performance is the same as when looking at all instances, but the relative improvements are much smaller.

The column *Standard Dijkstra calls* in Table 3 excludes these situations and hence shows an even larger gain than the column *All Dijkstra calls*. The question how to model rip-up costs efficiently when computing potentials remains for future research.

5.2 Reservations and discounts for incremental routing

In chip design practice, there are two main scenarios where a detailed routing is not computed from scratch, using just a global routing as input, but in an incremental way, using an approximate detailed routing as input. The first scenario is when a detailed routing has already been computed, but now a few changes have been made, for example in order to correct the logical function of the chip or to improve its timing behavior. The second scenario is when a step in between global and detailed routing is used, typically called track assignment, that maps the global wires to routing tracks in a way that obeys most—but not all—design rules.

In both scenarios of incremental routing, we get an almost feasible detailed routing as part of the input, and the task is to compute a completely feasible detailed routing by doing only few changes. While it is not exactly specified what "few" means, the motivation is that the input routing has already been optimized, for example with respect to the timing behavior of the chip; moreover, one aims at saving runtime.

The traditional approach to incremental routing is to check for violations of design rules (e.g., wires of different nets overlapping) and to try to repair such violations locally, in a relatively small area around that violation. While this can be parallelized very well, many violations cannot be repaired locally, and then the overall approach may fail or resort to global path searches as backup. Moreover, if the wiring of a net needs to be repaired in multiple places, the final result can be quite bad, for example with too many detours to meet timing constraints.

We suggest to repair violations globally but with a preference of using the initial solution. To this end, we convert any detailed wire in the input to a global wire and possibly a reservation. A reservation reserves that space for the particular net. When other nets are routed earlier, this space is blocked. Therefore, reservations are created

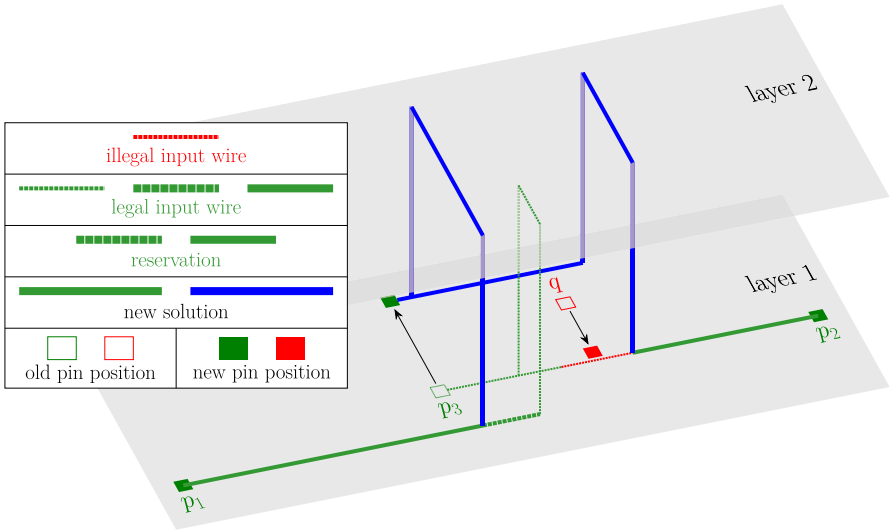

Fig. 9 Example of re-routing a net using reservations after changes to the input have been made. The green and red wires connected pins p_1, p_2, and the old position of p_3. Now suppose p_3 has been moved and is no longer connected. Moreover, pin q from a different net has been moved and now makes the red piece of wire illegal. Now we want to connect p_1, p_2, and the new position of p_3 and use much of the old wiring. We convert all of the old wiring (green and red) to global wires and add a global wire (not shown) that connects to the new position of p_3. Next we create reservations. Even though all of the green wires are legal, we may choose to create reservations only for the thick green wires, e.g., if we expect the harm of blocking other nets to outweigh the benefit of keeping them usable for this net. When this net is being routed, we may end up discarding the dashed part of the reservations, using the rest of the reservations and adding the blue wires. The solid wires then constitute the new routing for this net (color figure online)

only for (parts of) detailed wires that do not conflict with other detailed wires in the input. For an example situation, see Fig. 9.

Once a net is routed for which we have created reservations, we would like to encourage, but not force, the net to use the reserved space. We do this by defining a discount factor $0 < \delta < 1$ and multiplying all edge costs on a reservation of that net by δ. There are two reasons for using this incentive to route nets similarly as in the input: first, during detailed routing, we cannot do a complete timing analysis (this would be too slow), and the input routing has already been analyzed carefully. Second, we would hope for a speedup if the reservation serves as a useful guide how to route a net.

However, this speedup does not come automatically. In fact, with the traditional goal-oriented search techniques, reservations would lead to a slow-down. For example, if we define the potential π to be the ℓ_1-distance to the nearest target, multiplied by the minimum edge weight in that direction, then we have to multiply it by δ if there are any reservations in that direction (no matter how useful). Our generalized framework, however, allows us to refine the grid not only with respect to the targets, but also with respect to the reservations, and define individual (discounted) costs on the edges corresponding to reservations.

Table 4 Performance of different incremental routing algorithms when applied right after bulk routing on all nets

Chip	Algorithm	Preprocessing h:mm	Dijkstra Runtime h:mm	Dijkstra Labels 10^9	Total BonnRoute Wall time h:mm
A1	No reservations	0:58	3:55	2.7	0:23
A1	Reservations	1:04	2:23	0.5	0:15
A2	No reservations	0:42	4:19	3.6	0:19
A2	Reservations	0:50	2:26	0.6	0:13
A3	No reservations	0:40	4:12	3.5	0:18
A3	Reservations	0:45	2:06	0.6	0:12
B1	No reservations	4:17	23:18	17.6	1:12
B1	Reservations	4:31	8:52	3.3	0:42
B2	No reservations	16:43	87:29	58.5	4:12
B2	Reservations	15:58	54:17	14.8	3:11
B3	No reservations	9:19	40:29	35.3	2:51
B3	Reservations	10:32	16:46	6.3	2:24
C1	No reservations	1:00	70:57	59.4	2:06
C1	Reservations	1:27	25:47	9.3	1:34
C2	No reservations	22:56	2114:25	902.8	39:10
C2	Reservations	21:31	1140:54	309.6	23:35
D1	No reservations	8:17	1440:19	793.2	29:09
D1	Reservations	10:54	4 32:58	142.0	14:03
Sum	No reservations	64:56	3789:26	1877.1	79:44
Sum	Reservations	67:34	1686:34	487.4	46:14

Both runs use the general potential. The cost of reservations is multiplied by $\delta = \frac{3}{4}$

Table 5 Testbed consisting of six snapshots taken during a physical design flow used in production

Chip	Tech	l	$\lvert V\rvert$ 10^9	Area mm^2	Wires m	Vias 10^6	Nets 10^6	Pins 10^6	Calls 10^6	t	pql
b1	7 nm	16	5.7	0.40	6.6	4.22	0.41	0.48	0.04	27.95	16623
b2	7 nm	16	7.1	0.47	9.8	6.66	0.69	0.74	0.05	29.68	20696
b3	7 nm	16	5.4	0.36	10.0	7.03	0.72	0.75	0.04	35.91	29935
b4	7 nm	16	5.0	0.36	14.8	11.19	1.04	1.06	0.02	35.34	27754
b5	7 nm	16	6.6	0.46	15.2	13.80	1.26	1.27	0.01	30.64	21589
b6	7 nm	16	8.9	0.63	20.9	13.95	1.32	1.43	0.14	32.86	21524

Snapshots were taken after detailed routing based timing optimization, right before incremental detailed routing. For an explanation of the columns, see Table 2. Because the input already contains many valid connections, the number of calls to our Dijkstra implementation is often significantly smaller than the number of nets

Table 6 Performance of different incremental routing algorithms when applied after detailed routing based timing optimization on all paths that contain at least one illegal wire

Chip	Algorithm	Preprocessing h:mm	Dijkstra Runtime h:mm	Dijkstra Labels 10^9	Total BonnRoute Wall time h:mm
b1	No reservations	0:16	4:09	3.4	0:25
b1	Reservations	0:16	2:16	1.4	0:15
b2	No reservations	0:16	4:24	3.7	0:38
b2	Reservations	0:18	3:54	2.8	0:36
b3	No reservations	0:23	4:03	3.0	1:23
b3	Reservations	0:24	3:38	2.3	1:21
b4	No reservations	0:18	11:16	4.5	0:56
b4	Reservations	0:17	6:09	2.2	0:46
b5	No reservations	0:08	5:31	2.1	0:51
b5	Reservations	0:06	2:05	0.7	0:44
b6	No reservations	0:49	21:26	12.1	1:15
b6	Reservations	0:50	16:05	7.8	1:09
Sum	No reservations	2:12	50:51	29.0	5:30
Sum	Reservations	2:13	34:09	17.4	4:54

Both algorithms use the general potential. The cost of reservations is multipiled by $\delta = \frac{3}{4}$. For an explanation of the columns, see Table 3

If we use the output of track assignment as input, this often is a good solution on higher layers, where we have mostly longer wires, but much less so on lower layers, which are primarily used for pin access. (This is because long wires on low layers have a high resistance and thus poor delay.) In this case we may define reservations only on high layers and let the pin access and the short wires be freely determined by the detailed router.

To evaluate the effect of reservations on incremental routing, we compare two different algorithms. Both of them replace the same subset of the input wires by global wires and compute a new solution. What subset is chosen depends on the scenario and is described below. The algorithm *no reservations* is our standard bulk routing algorithm, starting from scratch using these global wires without any additional information. The algorithm *reservations* creates reservations for all input wires that are legal, except for short dangling wires that connect only to an illegal input wire. Both algorithms use the distance in the general model as their potential. In the algorithm *reservations*, the cost of reservations is multiplied by a discount factor of $\delta = \frac{3}{4}$ (i.e., 25% discount). The general model takes this discount into account.

We compare these two algorithms on 15 instances belonging to two different scenarios. In the first scenario, we start from an input in which all nets are connected and almost all wires are legal. More precisely, our input is a snapshot taken right after those bulk routing runs in Table 3 that used the general potential. We do not keep any part of our old solutions fixed, but replace all the detailed wires in nets we connected by global wires. The results of this experiment can be seen in Table 4.

The second scenario in which we evaluate the effect of reservations is a detailed routing that is no longer legal due to timing optimization. Table 5 gives an overview of this part of the testbed, consisting of six snapshots taken in a production flow just before incremental detailed routing. Unlike in the previous scenario, we keep pin-to-pin paths fixed if they consist only of detailed wires. Any wire not in such a path will be replaced by a global wire. See Fig. 9 for an example. The performance of both algorithms on these instances is shown in Table 6.

Acknowledgements We thank the many other contributors to BonnRoute, in particular Niko Klewinghaus, Christian Roth, and Niklas Schlomberg. Thanks also to Lukas Kühne, who started the initial implementation of the reservations concept. We also thank Niklas Schlomberg and the anonymous reviewers for carefully reading a preliminary version of our manuscript. Dorothee Henke has partially been supported by Deutsche Forschungsgemeinschaft (DFG) under Grant No. BU 2313/6, and the other authors under Grants EXC 59 and EXC-2047 (Hausdorff Center for Mathematics).

Funding Open Access funding enabled and organized by Projekt DEAL.

References

1. Ahrens, M.: Efficient algorithms for routing a net subject to VLSI design rules. Ph.D. thesis, University of Bonn (2020)
2. Ahrens, M., Gester, M., Klewinghaus, N., Müller, D., Peyer, S., Schulte, C., Téllez, G.: Detailed routing algorithms for advanced technology nodes. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **34**(4), 563–576 (2015)
3. Alpert, C.J., Mehta, D.P., Sapatnekar, S.S.: Handbook of Algorithms for Physical Design Automation. CRC Press (2008)
4. Batterywala, S., Shenoy, N., Nicholls, W., Zhou, H.: Track assignment: a desirable intermediate step between global routing and detailed routing. In: Proceedings of the 2002 IEEE/ACM International Conference on Computer-Aided Design , pp. 59–66 (2002)
5. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numer. Math. **1**(1), 269–271 (1959)
6. Edelsbrunner, H., Guibas, L., Stolfi, J.: Optimal point location in a monotone subdivision. SIAM J. Comput. **15**(2), 317–340 (1986)
7. Gester, M., Müller, D., Nieberg, T., Panten, C., Schulte, C., Vygen, J.: BonnRoute: algorithms and data structures for fast and good VLSI routing. ACM Trans. Des. Autom. Electron. Syst. **18**(2), 1–24 (2013)
8. Hart, P., Nilsson, N., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths. IEEE Trans. Syst. Sci. Cybern. **4**, 100–107 (1968)
9. Held, S., Müller, D., Rotter, D., Scheifele, R., Traub, V., Vygen, J.: Global routing with timing constraints. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **37**(2), 406–419 (2018)
10. Hetzel, A.: A sequential detailed router for huge grid graphs. In: Proceedings of Design, Automation and Test in Europe. IEEE, pp. 332–338 (1998)
11. Kirkpatrick, D.: Optimal search in planar subdivisions. SIAM J. Comput. **12**(1), 28–35 (1983)
12. Klewinghaus, N.: Efficient Detailed Routing on Optimized Tracks. Ph.D. thesis, University of Bonn (2022)

13. Lawler, E., Luby, M., Parker, B.: Finding shortest paths in very large networks. In: Nagl, M., Perl, J., Linz, T. (Eds), Proceedings of Graph-Theoretic Concepts in Computer Science (1983)
14. Lipton, H.J., Tarjan, R.E.: Applications of a planar separator theorem. In: 18th Annual IEEE Symposium on Foundations of Computer Science, pp. 162–170 (1977)
15. Müller, D., Radke, K., Vygen, J.: Faster min-max resource sharing in theory and practice. Math. Program. Comput. **3**(1), 1–35 (2011)
16. Peyer, S., Rautenbach, D., Vygen, J.: A generalization of Dijkstra's shortest path algorithm with applications to VLSI routing. J. Discrete Algorithms **7**(4), 377–390 (2009)
17. Preparata, F.P., Müller, D.E.: Finding the intersection of n half-spaces in time $O(n \log n)$. Theoret. Comput. Sci. **8**(1), 45–55 (1979)
18. Rubin, F.: The Lee path connection algorithm. IEEE Trans. Comput. **23**, 907–914 (1974)
19. Sarnak, N., Tarjan, R.: Planar point location using persistent search trees. Commun. ACM **29**(7), 669–679 (1986)
20. Sarrafzadeh, M., Lee, D.-T.: Restricted track assignment with applications. Int. J. Comput. Geom. Appl. **4**(1), 53–68 (1994)
21. Téllez, G., Hu, J., Wei, Y.: Routing. In: Lavagno, L., Markov, I.L., Martin, G., Scheffer, L.K. (Eds), Electronic Design Automation for IC Implementation, Circuit Design, and Process Technology. CRC Press (2016)