A Generalization of Dijkstra's Shortest Path Algorithm with Applications to VLSI Routing

Sven Peyer[1,2] **Dieter Rautenbach**[3] **Jens Vygen**[1]

Abstract

We generalize Dijkstra's algorithm for finding shortest paths in digraphs with non-negative integral edge lengths. Instead of labeling individual vertices we label subgraphs which partition the given graph. We can achieve much better running times if the number of involved subgraphs is small compared to the order of the original graph and the shortest path problems restricted to these subgraphs is computationally easy.

As an application we consider the VLSI routing problem, where we need to find millions of shortest paths in partial grid graphs with billions of vertices. Here, our algorithm can be applied twice, once in a coarse abstraction (where the labeled subgraphs are rectangles), and once in a detailed model (where the labeled subgraphs are intervals). Using the result of the first algorithm to speed up the second one via goal-oriented techniques leads to considerably reduced running time. We illustrate this with a state-of-the-art routing tool on leading-edge industrial chips.

1 Introduction

The shortest path problem is one of the most elementary, important and well-studied algorithmical problems [6, 13]. It appears in countless practical applications. The basic strategy for solving it in digraphs $G = (V(G), E(G))$ with non-negative edge lengths is Dijkstra's algorithm [10]. In its fastest strongly polynomial implementation using Fibonacci heaps [12] it has a running time of $O(|V(G)| \log |V(G)| + |E(G)|)$.

Various techniques have been proposed for speeding up Dijkstra's original labeling procedure. For undirected graphs a linear running time can be achieved for integral lengths [37] or on average in a randomized setting [14, 30]. For many instances that arise in practice, the graphs have some underlying geometric structure which can also be exploited to speed up shortest path computations [26, 36, 40]. Similarly, instances might allow a natural hierarchical decomposition or pre-processing might reveal local information which shortest path computations can use [2, 24, 31, 34, 35]. Goal-oriented techniques which use estimates or lower bounds such as the A^* algorithm have first been described as heuristics

[1]Forschungsinstitut für Diskrete Mathematik, Universität Bonn, Lennéstr. 2, D-53113 Bonn, Germany, e-mail: { peyer, vygen }@or.uni-bonn.de

[2]Corresponding author

[3]Institut für Mathematik, TU Ilmenau, Postfach 100565, D-98684 Ilmenau, Germany, e-mail: dieter.rautenbach@tu-ilmenau.de

in the artificial-intelligence setting [11, 16] and were later discussed as exact algorithms in the general context and in combination with other techniques [4, 5, 15, 23, 32]. There has been tremendous experimental effort to evaluate combinations of these techniques [9, 21]. See [41] for a comprehensive overview on speed-up techniques published during the last years.

The motivation for the research presented here originates from the routing problem in VLSI design where the time needed to complete the full design process is one of the most crucial issues to be addressed. In present day instances of this application millions of shortest paths have to be found in graphs with billions of vertices. Therefore, no algorithm which does not heavily exploit the specific instance structure can lead to an acceptable running time.

A simplified view of the VLSI routing problem is as follows. In a subgraph of a three-dimensional grid we look for vertex-disjoint Steiner trees connecting given terminal sets (nets). There are additional complications due to non-uniform wire widths and distance rules, pins (terminals) that are sets of (off-grid) shapes rather than single grid nodes, and extra rules for special nets. As these have little impact on the algorithmic core which is the subject of this paper, we do not discuss them here in detail (though our implementation within BonnRoute, a program that is used for routing many of the most complex industrial chips [25], takes all such constraints into account).

The standard approach is to compute a *global routing* [1, 22, 39] first, i.e. packing Steiner trees in a condensed grid subject to edge capacities. In a second step the *detailed routing* determines the actual layout, essentially by sequentially computing shortest paths, each connecting two components, within the global routing corridors. Heuristics like ripup-and-reroute allow to revise earlier decisions if one gets stuck. Some routers have an intermediate step (track assignment [3], congestion-driven wire planning [7]) between global and detailed routing or more than two coarsening levels [8], but we do not.

The core problem of essentially all detailed routers is to find a shortest wiring connection between two metal components that must be connected electrically. This can be modeled as a shortest path problem in a partial grid graph (see Section 3 for details). In contrast to many other applications in practice (e.g. finding shortest paths in road networks), where an expensive pre-processing of a fixed static graph is a reasonable and powerful approach to reduce the actual query time, the instance graph in the context of VLSI routing is different for each single path search.

While traditional routers use a very simple version of Dijkstra's algorithm (known as maze running or Lee's algorithm [27, 20]), several speed-up techniques are used routinely today. Some give up to find a shortest path (line search [19], line expansion [18], tile expansion [29]), but as longer paths waste space and are worse in terms of timing and power consumption, we do not consider this relaxation. Note that detours of the routing paths which are necessary due to congestion and capacity contraints and allowable due to sufficient timing margin are determined during global routing and are encoded in the global routing corridors. During global routing it is actually assumed that the final paths will be close to shortest paths within their corridor.

Among exact speed-up approaches, goal-oriented techniques [33] are widely used and

have been proven to be very efficient. Hetzel [17] proposed to represent the partial grid graph by a set of intervals of adjacent vertices and to label these intervals rather than individual vertices in his goal-oriented version of Dijkstra's algorithm. Part of our work is a generalization of Hetzel's algorithm. Cong et al. [7] and Li et al. [28] construct a connection graph, which is part of the Hanan grid induced by all obstacles. Xing and Kao [42] consider a similar graph but propagate piecewise linear functions on the edges of this graph. Our algorithm is also a generalization of this approach.

One of our main ideas is to introduce three levels of hierarchy. The vertices of the graph are the elements of the lowest and most detailed level. The middle level is a partition of the vertex set. Therefore, the elements of the middle level correspond to sets of vertices. Instead of labeling individual vertices as in the original version of Dijkstra's algorithm, we label the elements of this middle level. Finally, the top level is a partition of the middle level, i.e. several of the vertex sets of the middle level are associated. The role of the top level of the hierarchy is that it allows to delay certain labeling operations. We perform labeling operations between elements of the middle level that are contained in the same element of the top level instantly whereas all other labeling operations will be delayed. Depending on the structure of the underlying graph, the well-adapted choice of the hierarchy and the implementation of the labeling operations, we can achieve running time reductions both in theory and in practice.

The new algorithm which we call GeneralizedDijkstra provides a speed-up technique for Dijkstra's algorithm in two ways. First, it can directly be applied to propagate distance labels through a graph. The main difference to other techniques is that our approach labels sets of vertices instead of individual vertices. It is beneficial if vertices can be grouped such that their distances can be expressed by a relatively simple distance function and updating neighboring sets works fast. In our VLSI application, the vertex sets are one-dimensional intervals of the three-dimensional grid. A second application of GeneralizedDijkstra is the goal-oriented search. In the last few years, three main techniques to determine a good lower bound have been discussed in the literature: metric dependent distances, distances obtained by landmarks and distances from graph condensation [41]. The approach proposed in this paper is another method to compute lower bounds: It computes shortest distances from the target to all vertices in a supergraph G' of the reverse graph of the input graph. Here, G' must be chosen such that it approximately reflects the original distances and allows a partition of the vertex set in order to perform a fast propagation of distance labels. In our VLSI application, G' is the subgraph respresenting the global routing corridor, which is the union of only few rectangles.

In Section 2 we present a generic version of our algorithm. Section 3 is devoted to two applications of our strategy which lead to significant speed-ups for the described VLSI application. In Subsection 3.1 we show how to apply the algorithm in the case where the elements of the middle level induce two-dimensional rectangles in an underlying three-dimensional grid graph. In Subsection 3.2 we explain how our generic framework can be applied to detailed routing, generalizing a procedure proposed by Hetzel [17]. This time the elements of the middle level correspond to one-dimensional intervals. The experimental results in Section 4 show that we can reduce the overall running time of VLSI routing

significantly.

2 Generalizing Dijkstra's Algorithm

Throughout this section let $G = (V(G), E(G))$ be a digraph with non-negative integral edge lengths $c : E(G) \to \mathbb{Z}_{\geq 0}$. For vertices $u, v \in V(G)$ we denote by $\text{dist}_{(G,c)}(u, v)$ the minimum total length of a path in G from u to v with respect to c, or ∞ if v is not reachable from u. For a given non-empty source set $S \subseteq V(G)$ we define a function $d : V(G) \to \mathbb{Z}_{\geq 0} \cup \{\infty\}$ by

$$d(v) := \text{dist}_{(G,c)}(S, v) := \min\{\text{dist}_{(G,c)}(s, v) \mid s \in S\}$$

for $v \in V(G)$. If we are given a target set $T \subseteq V(G)$ we want to compute the distance

$$d(T) := \text{dist}_{(G,c)}(S, T) := \min\{d(t) \mid t \in T\}$$

from S to T in G with respect to c, or ∞ if T is not reachable from S.

Instead of labeling individual vertices with distance-related values, we label subgraphs of G induced by subsets of vertices with distance-related functions. Therefore, we assume to be given a set $\mathcal{V}$ of disjoint subsets of $V(G)$ and subsets $\mathcal{S}$ and $\mathcal{T}$ of $\mathcal{V}$ such that

$$V(G) = \dot{\bigcup_{U \in \mathcal{V}}} U \quad \text{and} \quad S = \dot{\bigcup_{U \in \mathcal{S}}} U \quad \text{and} \quad T = \dot{\bigcup_{U \in \mathcal{T}}} U.$$

We require that the graph $\mathcal{G}$ with $V(\mathcal{G}) := \mathcal{V}$ and

$$E(\mathcal{G}) := \{(U, U') \mid \exists u \in U, u' \in U', (u, u') \in E(G) \text{ with } c((u, u')) = 0\}$$

is acyclic. (Note that we do not need to assume this for G. Moreover, one can always get this property by contracting strongly connected components of $(V(G), \{e \in E(G) : c(e) = 0\}$.) Therefore, there is a topological order $V_1, V_2, \ldots, V_{|\mathcal{V}|}$ of $\mathcal{V}$ with $i < j$ if $(V_i, V_j) \in E(\mathcal{G})$. For $U \in \mathcal{V}$ we define the *index* of U to be $I(U) = i$ iff $U = V_i$.

Throughout the execution of the algorithm and for every $U \in \mathcal{V}$ we maintain a function $d_U : U \to \mathbb{Z}_{\geq 0} \cup \{\infty\}$ which is an upper bound on d, i.e.

$$d_U(v) \geq d(v) \text{ for all } v \in U, \tag{1}$$

and a feasible potential on $G[U]$, i.e.

$$d_U(v) \leq d_U(u) + c((u, v)) \text{ for all } (u, v) \in E(G[U]), \tag{2}$$

where $G[U]$ denotes the subgraph of G induced by U.

Initially, we set

$$d_U(v) := \begin{cases} 0 & \text{for } v \in U \in \mathcal{S}, \\ \infty & \text{for } v \in U \in \mathcal{V} \setminus \mathcal{S}. \end{cases}$$

We want to make use of a specific structure of the graph $\mathcal{G}$ and distinguish between two different labeling operations. For this, we additionally require a partition of $\mathcal{V}$ into $N \geq 1$ sets $\mathcal{V}_1, \ldots, \mathcal{V}_N$, called *blocks*, and a function $\mathcal{B} : \mathcal{V} \to \{\mathcal{V}_1, \ldots, \mathcal{V}_N\}$ such that

$$\begin{aligned} \forall\, 1 \leq i \leq N: &\quad \emptyset \neq \mathcal{V}_i \subseteq \mathcal{V}, \\ \forall\, U \in \mathcal{V}: &\quad U \in \mathcal{B}(U), \\ \forall\, 1 \leq i < j \leq N: &\quad \mathcal{V}_i \cap \mathcal{V}_j = \emptyset. \end{aligned}$$

Clearly, $\mathcal{V} = \bigcup_{i=1}^{N} \mathcal{V}_i$ and $V(G) = \bigcup_{i=1}^{N} \bigcup_{U \in \mathcal{V}_i} U$ by the definition of blocks.

A central idea of our algorithm GeneralizedDijkstra is the following: We distinguish between two operations for a vertex set $U \in \mathcal{V}$ which is chosen to label its neighbors: U *directly updates* the neighboring sets within the same block and *registers* labeling operations to vertex sets in different blocks for a later use. This approach has two advantages: First, many registered labeling operations may never have to be performed if a target vertex is reached before the registered operations would be processed. Second, if sets in $\mathcal{V}$ typically have few of their neighboring sets within the same block, update operations between blocks may be much more efficient when performed at once instead of one after another. For a schematic illustration of our algorithm see Figure 1. Two more examples are given in Section 3.

Our algorithm maintains a function $\text{key} : \mathcal{V} \to \mathbb{Z}_{\geq 0} \cup \{\infty\}$ and a queue $\mathcal{Q} = \{U \in \mathcal{V} \mid \text{key}(U) < \infty\}$, allowing operations to insert an element, to decrease the key of an element and to delete an element of minimum key. At any stage in the algorithm, for each $U \in \mathcal{V}$, $\text{key}(U)$ is the minimum distance label of any vertex in U that was decreased after the last time that U was deleted from $\mathcal{Q}$. After U has updated its neighbors or registered a labeling operation, $\text{key}(U)$ is set to infinity. It can be reset to a value smaller than infinity, as soon a $d(u)$ is reduced for a vertex of $u \in U$ by an update operation onto U.

For two sets $\mathcal{U}, \mathcal{U}' \subseteq \mathcal{V}$ and a queue $\mathcal{Q} \subseteq \mathcal{V}$ we use the following update operation which clearly maintains (1) and (2):

Update$(\mathcal{U} \to \mathcal{U}', \mathcal{Q})$:
1 for all $U' \in \mathcal{U}'$ do:
2 for all $v \in U'$ do:
3 Set $\delta := \min_{U \in \mathcal{U}} \min_{u \in U} \left\{ d_U(u) + \text{dist}_{(G[\{u\} \cup U'], c)}(u, v) \right\}$.
4 if $\delta < d_{U'}(v)$ then
5 Set $d_{U'}(v) := \delta$.
6 Set $\text{key}(U') := \min\{\text{key}(U'), \delta\}$.
7 Set $\mathcal{Q} := \mathcal{Q} \cup \{U'\}$.

The actual labeling operation is done by Update. For each vertex set $U' \in \mathcal{U}'$ and each vertex $v \in U'$ it computes the minimum distance in the subgraph determined by U' and all neighboring vertices in vertex sets of $\mathcal{U}$. If the distance label at v can be decreased, the label of v and the key of U' will be updated. If U' is not in the queue, it will enter it. Of course, this is not done sequentially for each single vertex. Here, the advantage of our

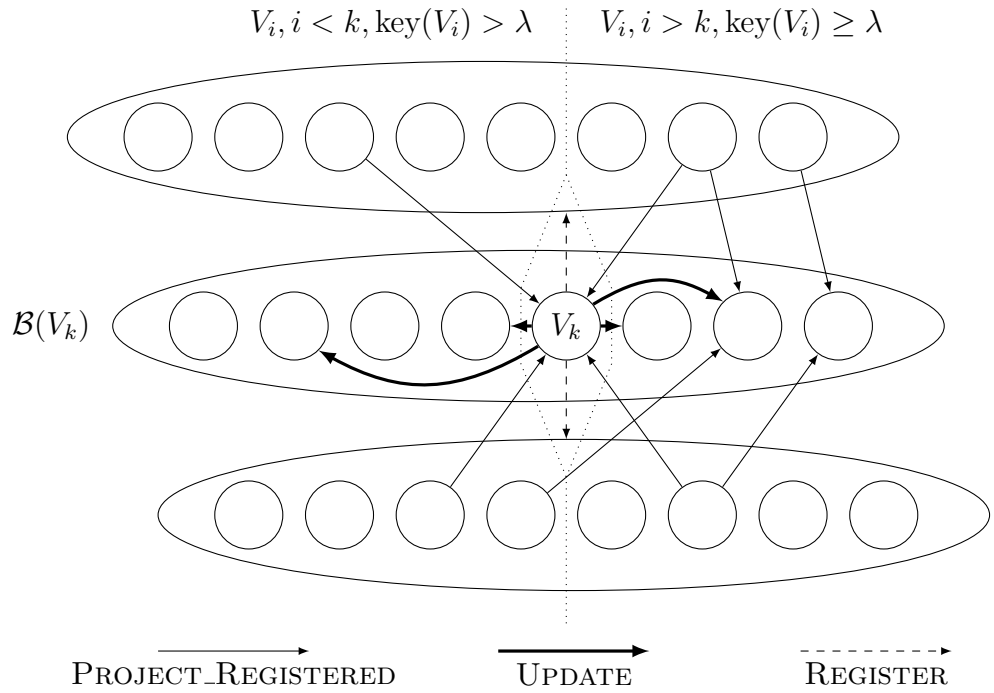

Figure 1: Schematic view of vertex sets V_i (circles) and blocks $\mathcal{V}_i$ (ellipses). The left-to-right order of the vertex sets is a topological order of $\mathcal{G}$. The arcs show update operations. If V_k and $\mathcal{B}(V_k)$ are selected in step 7 of GENERALIZED DISKSTRA, then $\text{key}(V_i) > \lambda$ and $\mathcal{R}(V_i, \lambda) = \emptyset$ for $i = 1, \ldots, k-1$, and this property is maintained. Then first all registered updates onto block $\mathcal{B}(V_k)$ are performed (PROJECT_REGISTERED, thin arcs), then the elements of $\mathcal{B}(V_k)$ with key λ are scanned in their order (we show V_k only), updates within the block are performed directly (UPDATE, bold arcs), and updates to other blocks are registered (REGISTER, dashed arcs). Each V_k is chosen at most once in phase λ.

algorithm becomes apparent: Instead of performing the labeling steps on a vertex by vertex basis, it rather updates the distance function of neighboring vertex sets in one step. The more carefully a partition of V is chosen and the simpler d_U becomes, the faster UPDATE can work. In our main algorithm, UPDATE is called for a single vertex set $\mathcal{U} := \{U\}$ updating its neighbors in $\mathcal{B}(U)$, and for a set of vertex sets with registered labels updating their neighbors in one block.

The second operation is the registration of labeling operations to be postponed. For this, we define a set $\mathcal{R}(\mathcal{U}, \lambda)$ for each block $\mathcal{U} \in \{\mathcal{V}_1, \ldots, \mathcal{V}_N\}$ and $\lambda \in \mathbb{Z}_{\geq 0}$, which consists of all vertex sets which might cause a label of value λ in some vertex set in $\mathcal{U}$. This set is filled by the following routine, where $E_G(U, U') := \{(u, u') \in E(G) \mid u \in U, u' \in U'\}$ and $\min \emptyset := \infty$.

REGISTER($U \to \mathcal{U}', \mathcal{R}$):
1 Set $\lambda' := \min_{U' \in \mathcal{U}'} \min \{\delta \mid \delta = d_U(u) + c((u, v)) < d_{U'}(v), (u, v) \in E_G(U, U')\}$.
2 IF $\lambda' \neq \infty$ THEN:

3 Set $\mathcal{R}(\mathcal{U}', \lambda') := \mathcal{R}(\mathcal{U}', \lambda') \cup \{U\}$.

REGISTER is called for a vertex set U and some block $\mathcal{U}' \neq \mathcal{B}(U)$. It computes the minimum label λ' which improves a label of at least one vertex in a neighboring vertex set of U in $\mathcal{U}'$ and registers U in $\mathcal{R}(\mathcal{U}', \lambda')$. If no label can be decreased, U will not be registered.

Given a block $\mathcal{U}$ and a key λ, we apply two major subroutines: First, all labeling operations of registered vertex sets of $\mathcal{U}$ at value λ will be performed onto block $\mathcal{U}$ in PROJECT_REGISTERED. Afterwards, all vertex sets in $\mathcal{U}$ containing vertices with key λ update their neighbors within the same block and register labeling operations in different blocks in PROJECT_FROMBLOCK. We will use the notation $\mathcal{Q}_\lambda := \{U \in \mathcal{Q} \mid \text{key}(U) = \lambda\}$ for $\lambda \in \mathbb{Z}_{\geq 0}$.

PROJECT_REGISTERED$(\mathcal{U}, \lambda, \mathcal{Q}, \mathcal{R})$:
1 IF $\mathcal{R}(\mathcal{U}, \lambda) \neq \emptyset$ THEN:
2 UPDATE$(\mathcal{R}(\mathcal{U}, \lambda) \to \mathcal{U}, \mathcal{Q})$.
3 Set $\mathcal{R}(\mathcal{U}, \lambda) := \emptyset$.

PROJECT_FROMBLOCK$(\mathcal{U}, \lambda, \mathcal{Q}, \mathcal{R})$:
1 WHILE there is an element $U \in \mathcal{Q}_\lambda \cap \mathcal{U}$ DO:
2 Choose $U \in \mathcal{Q}_\lambda$ with minimum index.
3 Set $\mathcal{Q} := \mathcal{Q} \setminus \{U\}$ and $\text{key}(U) := \infty$.
4 IF $U \in \mathcal{T}$ THEN:
5 RETURN λ.
6 FOR ALL $\mathcal{U}' \in \{\mathcal{B}(U') \mid U' \in \mathcal{V} \setminus \{U\}$ with $E_G(U, U') \neq \emptyset\}$ DO:
7 IF $\mathcal{U}' = \mathcal{U}$ THEN:
8 Set $\mathcal{J}_U := \{U' \in \mathcal{U} \setminus \{U\} \mid E_G(U, U') \neq \emptyset\}$.
9 UPDATE$(\{U\} \to \mathcal{J}_U, \mathcal{Q})$.
10 ELSE
11 REGISTER$(U \to \mathcal{U}', \mathcal{R})$.

PROJECT_REGISTERED makes up for all postponed labeling steps onto block $\mathcal{U}$ at label λ. After this, $\mathcal{R}(\mathcal{U}, \lambda)$ is empty. PROJECT_FROMBLOCK goes over all vertex sets $U \in \mathcal{U}$ in the queue whose key equals the current label λ according to their topological order. If a target vertex set is the minimum element in the queue, the overall algorithm stops. Otherwise, all neighboring vertex sets in the same block are directly labeled by UPDATE whereas labeling operations to neighbors in different blocks are registered by REGISTER.

Finally, we can formulate the overall algorithm. It performs labeling operations as long as no set in $\mathcal{T}$ has received its final label and there are still labeling operations which need to be executed. It runs in so-called *phases* where in the phase at key λ all vertices with distance λ will receive their final label. In phase with key λ, the block $\mathcal{U}$ is chosen which includes a vertex set of minimum index containing a vertex with key λ. If no such vertex exists, a block $\mathcal{U}$ with postponed labels of value λ is taken. For $\mathcal{U}$, PROJECT_REGISTERED and

PROJECT_FROMBLOCK are called in that order. PROJECT_REGISTERED must be called before labeling steps from vertices in vertex sets of $\mathcal{U}$ are performed within $\mathcal{U}$ in order to update vertices at label λ by neighbors of different blocks. Otherwise, PROJECT_FROMBLOCK might be not operate efficiently because a vertex might get a label λ at a later time in the algorithm which again requires an update operation for neighbors in $\mathcal{U}$. This loop at key λ is done as long as their is still a vertex to be scanned or a block with postponed labels. The algorithm stops as soon as a vertex in $\bigcup \mathcal{T}$ receive its final label and returns it, or it returns infinity to indicate that $\bigcup \mathcal{T}$ is not reachable.

GENERALIZEDDIJKSTRA$(G, c, \mathcal{V}, \mathcal{B}, \mathcal{S}, \mathcal{T})$:

1 Set key$(U) := 0$ and $d_U(v) := 0$ for $U \in \mathcal{S}$ and $v \in U$.
2 Set key$(U) := \infty$ and $d_U(v) := \infty$ for $U \in \mathcal{V} \setminus \mathcal{S}$ and $v \in U$.
3 Set $\mathcal{R}(\mathcal{U}, \lambda) := \emptyset$ for $\mathcal{U} \in \{\mathcal{V}_1 \ldots, \mathcal{V}_N\}$ and $\lambda \in \mathbb{Z}_{\geq 0}$.
4 Set $\mathcal{Q} := \mathcal{S}$ and $\lambda := 0$.
5 WHILE $\mathcal{Q} \neq \emptyset$ or $\mathcal{R}(\mathcal{U}, \lambda') \neq \emptyset$ for some $\lambda' \geq \lambda$ and some $\mathcal{U} \in \{\mathcal{V}_1, \ldots, \mathcal{V}_N\}$ DO:
6 WHILE $\mathcal{Q}_\lambda \neq \emptyset$ or $\mathcal{R}(\mathcal{U}, \lambda) \neq \emptyset$ for some $\mathcal{U} \in \{\mathcal{V}_1, \ldots, \mathcal{V}_N\}$ DO:
7 Choose $\mathcal{U} \in \{\mathcal{V}_1, \ldots, \mathcal{V}_N\}$ s.t. $\arg\min\{I(U) \mid U \in \mathcal{Q}_\lambda$ or $\mathcal{R}(\mathcal{U}, \lambda) \neq \emptyset\} \in \mathcal{U}$.
8 PROJECT_REGISTERED$(\mathcal{U}, \lambda, \mathcal{Q}, \mathcal{R})$.
9 PROJECT_FROMBLOCK$(\mathcal{U}, \lambda, \mathcal{Q}, \mathcal{R})$.
10 Set $\lambda := \min\{\mu : \mathcal{Q}_\mu \neq \emptyset$ or $\mathcal{R}(\mathcal{U}, \mu) \neq \emptyset$ for some $\mathcal{U} \in \{\mathcal{V}_1, \ldots, \mathcal{V}_N\}\}$.
11 RETURN ∞.

Theorem 1. *The algorithm* GENERALIZEDDIJKSTRA *calculates the correct distance from* $S := \bigcup \mathcal{S}$ *to* $T := \bigcup \mathcal{T}$. *If there is no path from* S *to* T, *the algorithm computes the minimum distance from* S *to all reachable vertices in* G *and returns* ∞.

Proof: We first show that for every $\lambda \in \mathbb{Z}_{\geq 0}$

$$\bigcup_{U \in \mathcal{V}} \{v \in U \mid d_U(v) = \lambda\} = \bigcup_{U \in \mathcal{V}} \{v \in U \mid d(v) = \lambda\} \tag{3}$$

holds after execution of phase λ which ends when λ is increased in line 10 of GENERALIZEDDIJKSTRA. For contradiction, assume that there is a λ for which equation (3) does not hold. Choose λ minimum possible. By (1), $\lambda = d(v) < d_U(v)$ for some $U \in \mathcal{V}$ and $v \in U$. We choose $U \in \mathcal{V}$ with $I(U)$ minimum possible, and $v \in U$ and a shortest path P from S to v such that $|E(P)|$ is minimum. Let u be the predecessor of v on P. Hence, $\lambda' := d(u) \leq \lambda$ and, by the choice of v, $d_{U'}(u) = \lambda'$ for $U' \in \mathcal{V}$ with $u \in U'$. We will show

$$d_U(v) \leq d_{U'}(u) + c((u, v)). \tag{4}$$

It directly follows for $U = U'$ by (2).

For $U \neq U'$, let $\mathcal{U} := \mathcal{B}(U)$ and $\mathcal{U}' := \mathcal{B}'(U)$. If $\lambda' < \lambda$, then U must have been updated directly from U' (if $\mathcal{U} = \mathcal{U}'$) or registered in PROJECT_FROMBLOCK$(\mathcal{U}', \lambda', \mathcal{Q}, \mathcal{R})$ (if $\mathcal{U} \neq \mathcal{U}'$) in phase λ'. In the latter case, d_U is updated by UPDATE$(\mathcal{R}(\mathcal{U}, \lambda) \to \mathcal{U}, \mathcal{Q})$ in

PROJECT_REGISTERED$(\mathcal{U}, \lambda, \mathcal{Q}, \mathcal{R})$. If $\lambda' = \lambda$, then $c((u,v)) = 0$. In this case, $I(U') < I(U)$ and U' must have been removed from $\mathcal{Q}$ before U (line 2 of PROJECT_FROMBLOCK). Consequently, U has been directly updated by U' in PROJECT_FROMBLOCK$(\mathcal{U}', \lambda, \mathcal{Q}, \mathcal{R})$ (if $\mathcal{U} = \mathcal{U}'$), or U' was registered in $\mathcal{R}(\mathcal{U}, \lambda)$ and has updated its neighbored vertex set U in PROJECT_REGISTERED$(\mathcal{U}, \lambda, \mathcal{Q}, \mathcal{R})$. For $\lambda' < \lambda$ as well as for $\lambda' = \lambda$, we conclude

$$d_U(v) \leq d_{U'}(u) + \text{dist}_{(G[\{u\} \cup U'], c)}(u, v) \leq d_{U'}(u) + c((u,v)),$$

proving inequality (4) for $U \neq U'$. By (4) and our assumption on v and u,

$$d_U(v) \leq d_{U'}(u) + c((u,v)) = d(u) + c((u,v)) = d(v) < d_U(v),$$

which is a contradiction. This concludes the proof of (3).

All phases with key less than λ have been finished already when phase λ is being processed. By (3), a vertex $v \in U$ with $d(v) < \lambda$ cannot get a distance label $d_U(v) = \lambda$ after phase $\lambda - 1$.

It follows that

$$\{v \in U : d_U(v) = \lambda\} \subseteq \{v \in U : d(v) = \lambda\} \tag{5}$$

holds after a vertex set U has been removed from Q at key λ.

Therefore, an element $T \in \mathcal{T}$ will be removed from the queue at minimum distance $\lambda = d(T)$ if T is reachable from S. Otherwise, GENERALIZEDDIJKSTRA stops and returns ∞ as soon as $\mathcal{Q}$ is empty and there are no labeling registrations left in $\mathcal{R}$. By (3), the distance from S to any vertex $v \in V(G)$ that is reachable from S is given by $d_U(v)$ for $v \in U \in \mathcal{V}$. □

In addition to the computed distance from S to T or from S to all vertices one is often also interested in shortest paths. These can be derived easily from the output of GENERALIZEDDIJKSTRA. Alternatively, one could store predecessor information during the shortest path computation encapsulated in UPDATE.

The theoretical running time of GENERALIZEDDIJKSTRA as well as its performance in practice depends on the structure of the underlying graph G and its partition into vertex sets and blocks. In the special case of $N = 1$, the running time reduces to

$$O\left((\Lambda + 1)(|\mathcal{V}| \log |\mathcal{V}| + \phi |E(\mathcal{G})|)\right),$$

where Λ is the length of a shortest path from S to T and ϕ is the running time of lines 2 and 3 of UPDATE. If $N = 1$ and all vertex sets are singletons, GENERALIZEDDIJKSTRA equals Dijkstra's algorithm with a running time of $O\left(|V(G)| \log |V(G)| + |E(G)|\right)$. The essential difference is that, for general vertex sets, an element of the queue can enter the queue again if it contains more than one vertex. Consequently, there are at most $\Lambda + 1$ many queries for the smallest element of the queue. Both time bounds assume that $\mathcal{Q}$ is implemented by a Fibonacci heap [12].

GENERALIZEDDIJKSTRA is primarily suitable for graphs with a regular structure, for which the ground set V can be partitioned into a set $\mathcal{V}$ of vertex sets with easily computable functions d_U for all $U \in \mathcal{V}$, e.g. linear functions. In the following section we will

describe two applications of GeneralizedDijkstra in VLSI routing where the algorithm is particularly efficient.

3 Applications in VLSI Routing

The graph typically used for modeling the search space for VLSI routing is a three-dimensional grid graph. The routing is realized in a small number of different *layers*, i.e. the number of different z-coordinates is very small (~ 10 presently) compared to the extension of the x-y-layers ($\sim 10^5 \times 10^5$ presently). Each x-y-layer is assigned a *preference direction* (x or y) and consecutive layers have different preference directions.

Let $G_0 := (V(G_0), E(G_0))$ be the infinite three-dimensional grid graph with vertex set $V(G_0) := \mathbb{Z}^3$, in which two vertices $(x, y, z) \in V(G_0)$ and $(x', y', z') \in V(G_0)$ are joined by a pair of oppositely directed edges of equal length if and only if $|x-x'|+|y-y'|+|z-z'| = 1$. Wires running orthogonally to the preference direction within one layer may block many wires in preference direction and should therefore be largely avoided. Moreover, *vias* (edges connecting adjacent layers) are undesired for several reasons, including their high electrical resistance and impact to the manufacturing yield. This is typically modeled by edge lengths $c : E(G_0) \to \mathbb{Z}_{\geq 0}$ that prefer edges within one layer in preference direction: for each layer $z \in \mathbb{Z}$ there are constants $c_{z,1}, c_{z,2}, c_z \in \mathbb{Z}_{>0}$ such that

$$\begin{aligned} c(((x,y,z),(x+1,y,z))) &= c_{z,1}, \\ c(((x,y,z),(x,y+1,z))) &= c_{z,2} \quad \text{and} \\ c(((x,y,z),(x,y,z+1))) &= c_z \end{aligned}$$

for all $x, y \in \mathbb{Z}$. Typical values used in practice and throughout this paper are 1 and 4 for edges within one layer in and orthogonal to the preference direction and 13 for vias. Figure 2 shows an instance of the detailed routing problem and its solution.

3.1 Labeling Rectangles

For speeding up the computation of a shortest path in a subgraph of G_0 using a goal-oriented approach one can use a good lower bound on the distances. In order to determine this lower bound, we consider just the corridor computed by global routing and neglect obstacles and previously determined paths intersecting it. Since these corridors are produced by global routing as a union of relatively few rectangles GeneralizedDijkstra computes distances efficiently. In this first application we use only one block, i.e. $N = 1$. Consequently, there are no registrations and Project_Registered will not be called.

Let $G = (V(G), E(G))$ be a subgraph of the infinite graph G_0 induced by a set $\mathcal{V}$ of *rectangles*, where a rectangle is a set of the form

$$[x_1, x_2] \times [y_1, y_2] \times \{z_1\} \;:=\; \big\{(x,y,z) \in \mathbb{Z}^3 \mid x_1 \leq x \leq x_2,\, y_1 \leq y \leq y_2,\, z = z_1\big\}$$

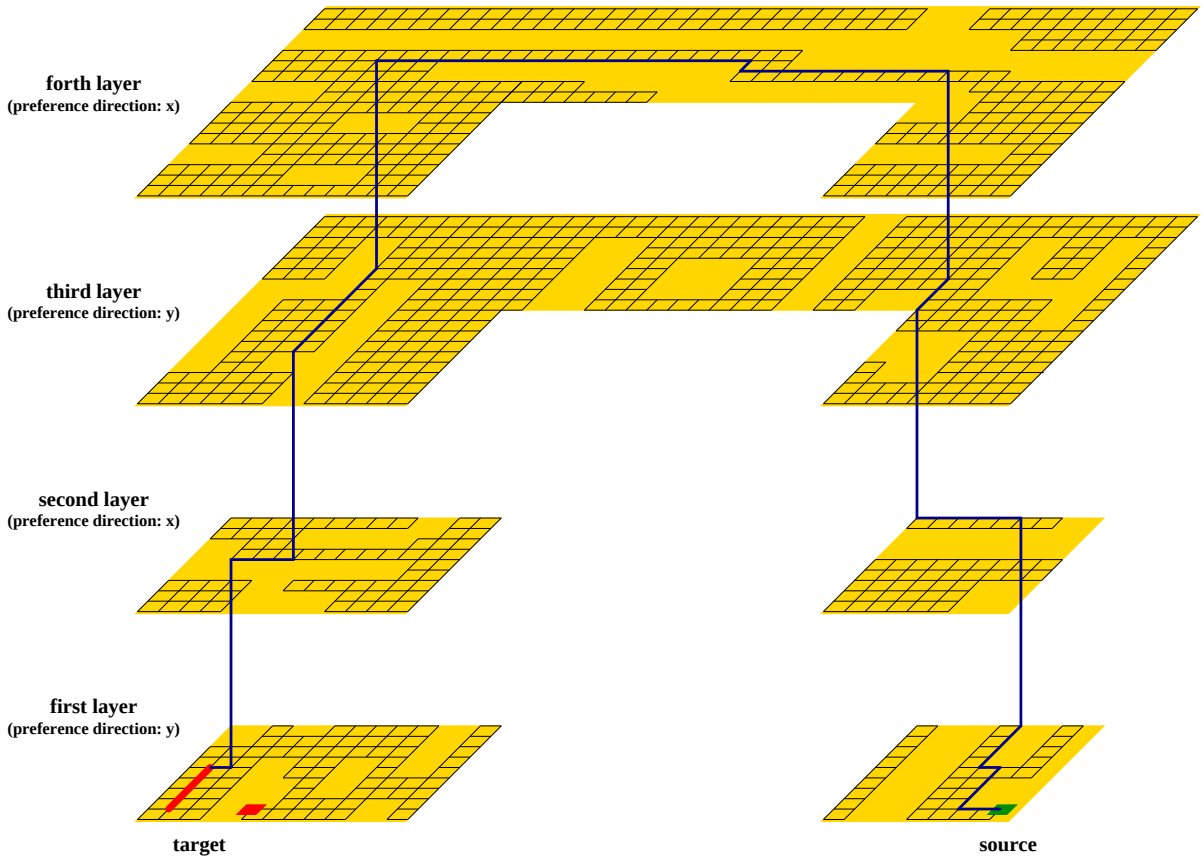

Figure 2: The source vertex (green) on the lower right corner and the target vertices (red) on the lower left corner of the picture are connected by a shortest path (blue) of length 153. The corridor determined by global routing (yellow) runs over four different layers in this example. The costs of edges running in and orthogonal to the preference direction are 1 and 4, resp., the cost of a via is 13.

for integers $x_1 \leq x_2$, $y_1 \leq y_2$ and z_1. Note that $x_1 = x_2$ or $y_1 = y_2$ is allowed, in which case the rectangles are intervals or just single points. Two rectangles R and R' are said to be *adjacent* if $G_0[R \cup R']$ is connected.

We assume that $\mathcal{V}$ satisfies the following condition which can always be ensured by iteratively splitting rectangles without creating many new rectangles for typical VLSI instances: For every two rectangles $R, R' \in \mathcal{V}$ with $R = [x_1, x_2] \times [y_1, y_2] \times \{z_1\}$ and $R' = [x'_1, x'_2] \times [y'_1, y'_2] \times \{z'_1\}$ we have that either $x_1 > x'_2$ or $x_2 < x'_1$ or $(x_1, x_2) = (x'_1, x'_2)$, and similarly, either $y_1 > y'_2$ or $y_2 < y'_1$ or $(y_1, y_2) = (y'_1, y'_2)$. Clearly, this condition implies that each rectangle has at most six adjacent rectangles. As an example, Figure 3 shows the partition of the routing area of the instance in Figure 2 into rectangles.

The set $Z := \{z \in \mathbb{Z} \mid \exists\, x, y \in \mathbb{Z} \text{ with } (x, y, z) \in V(G)\}$ contains all relevant z-coordinates, and the sets $C_i := \{0\} \cup \{c \in \mathbb{Z} \mid |c| = c_{z,i} \text{ for some } z \in Z\}$ contain all

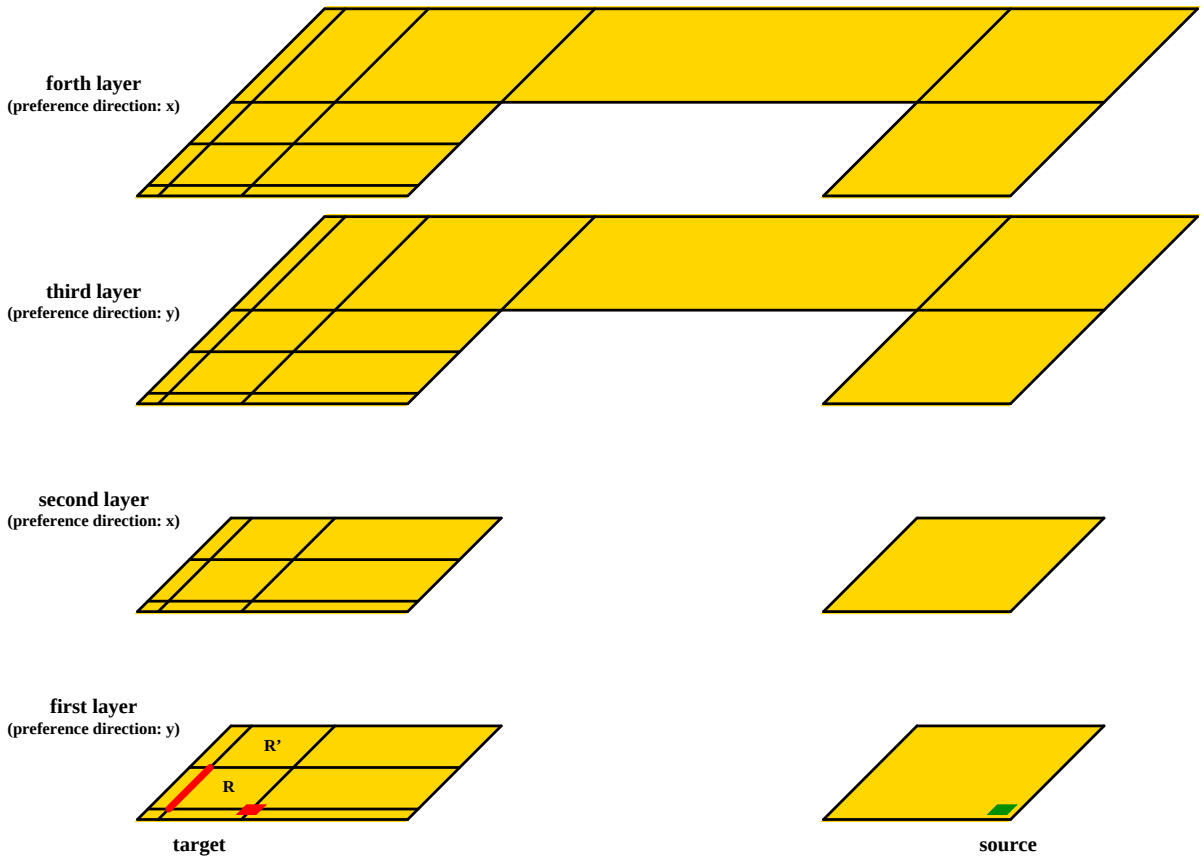

Figure 3: The corridor of the global routing for the instance depicted in Figure 2 is partitioned into rectangles to propagate distance functions. As an example, the function d_R of the rectangle R containing all target vertices at its boundary is given by $\min(4(x-x_1), -4(x-x_1)+(y-y_1)+16)$. The function $d_{R'}$ of the adjacent rectangle R' is $\min(4(x-x'_1)+(y-y'_1), -4(x-x'_1)+(y-y'_1)+20)$, where $x'_1 = x_1$ and $y'_1 = y_1 + 4$.

relevant edge lengths in x-direction ($i = 1$) and y-direction ($i = 2$), respectively. Let $k_i := |C_i|$ for $i = 1, 2$.

Due to the bounded number of different edge lengths, it is possible to store the function d_R corresponding to the function d_U of Section 2 implicitly as a minimum over $k_1 k_2$ linear functions assigned to each rectangle R:

$$d_R((x, y, z)) := \min\{d_{(R,c_1,c_2)}(x, y) \mid (c_1, c_2) \in C_1 \times C_2\},$$

where with $(c_1, c_2) \in C_1 \times C_2$ we associate a linear function $d_{(R,c_1,c_2)} : \mathbb{Z}^2 \to \mathbb{Z}_{\geq 0} \cup \{\infty\}$ of the form

$$d_{(R,c_1,c_2)}(x, y) = c_1(x - x_1) + c_2(y - y_1) + \delta_{(R,c_1,c_2)}.$$

See Figure 3 for examples.

All information about $d_{(R,c_1,c_2)}$ is contained in the offset value $\delta_{(R,c_1,c_2)}$. Initially,

$$\delta_{(R,c_1,c_2)} := \begin{cases} 0 & \text{for } (R,c_1,c_2) \in \mathcal{S} \times \{0\} \times \{0\}, \\ \infty & \text{for } (R,c_1,c_2) \in (\mathcal{V} \setminus \mathcal{S}) \times \{0\} \times \{0\}. \end{cases}$$

During the execution of the algorithm these values will be updated as follows: Let $(R,c_1,c_2) \in \mathcal{V} \times C_1 \times C_2$, and let $R' \in \mathcal{V} \setminus \{R\}$ be adjacent to R. For $(x',y',z') \in R'$ let

$$\begin{aligned} d_{((R,c_1,c_2)\to R')}(x',y') \;:=\; & \min\{d_{(R,c_1,c_2)}(x,y) + \mathrm{dist}_{(G_0[R\cup R'],c)}((x,y,z),(x',y',z')) \\ & \mid (x,y,z) \in R\}. \end{aligned}$$

The main observation established by the next lemma is that the function $d_{((R,c_1,c_2)\to R')}$ is of the same form as $d_{(R',c_1',c_2')}$ for appropriate values of c_1' and c_2'.

Lemma 2. *If $R \in \mathcal{V}$ and $R' = [x_1',x_2'] \times [y_1',y_2'] \times \{z'\} \in \mathcal{V}$ are adjacent, and $(c_1,c_2) \in C_1 \times C_2$, then there are $c_1' \in C_1$, $c_2' \in C_2$ and $\delta' \in \mathbb{Z}_{\geq 0}$ such that*

$$d_{((R,c_1,c_2)\to R')}(x',y') = c_1'(x'-x_1') + c_2'(y'-y_1') + \delta'$$

for all $(x',y',z') \in R'$.

Proof: We give details for just one case, since the remaining cases can be proved using similar arguments. Therefore, we assume that $R = [x_1,x_2] \times [y_1,y_2] \times \{z\}$ and $R' = [x_2+1,x_3] \times [y_1,y_2] \times \{z\}$ For $(x,y,z) \in R$ and $(x',y',z) \in R'$ we have

$$\mathrm{dist}_{(G_0[R\cup R'],c)}((x,y,z),(x',y',z)) = c_{z,1}|x-x'| + c_{z,2}|y-y'|.$$

Since $x < x'$, this simplifies to $c_{z,1}(x'-x)+c_{z,2}(y'-y)$ for $y \leq y'$ and $c_{z,1}(x'-x)-c_{z,2}(y'-y)$ for $y \geq y'$. Hence, by definition, $d_{((R,c_1,c_2)\to R')}(x',y')$ equals

$$\begin{aligned} \min \; \Bigg\{ & \min_{x_1\leq x\leq x_2} \min_{y_1\leq y\leq y'} \left(c_{z,1}(x'-x) + c_{z,2}(y'-y) + c_1(x-x_1) + c_2(y-y_1) + \delta_{(R,c_1,c_2)}\right), \\ & \min_{x_1\leq x\leq x_2} \min_{y'\leq y\leq y_2} \left(c_{z,1}(x'-x) - c_{z,2}(y'-y) + c_1(x-x_1) + c_2(y-y_1) + \delta_{(R,c_1,c_2)}\right) \Bigg\}. \end{aligned}$$

Depending on the signs of $(c_1-c_{z,1})$, $(c_2-c_{z,2})$ and $(c_2+c_{z,2})$, this minimum is attained — independently of the specific value of (x',y') — by setting

$$x := \begin{cases} x_1 & \text{if } c_1 - c_{z,1} \geq 0, \\ x_2 & \text{if } c_1 - c_{z,1} < 0 \end{cases} \qquad \text{and} \qquad y := \begin{cases} y_1 & \text{if } c_2 - c_{z,2} \geq 0, \\ y' & \text{if } c_2 - c_{z,2} < 0 \text{ and } c_2 + c_{z,2} \geq 0, \\ y_2 & \text{if } c_2 + c_{z,2} < 0 \end{cases}$$

and the desired result follows. □

This lemma shows that we can store d_R implicitly as a vector with k_1k_2 entries $\delta_{(R,c_1,c_2)}$ and that a labeling operation from one rectangle R to another R' can be done by manipulating the entries of the vector corresponding to $d_{R'}$. This can be done in constant time regardless of the cardinalities of R and R'.

We can now apply GeneralizedDijkstra implementing Update with the following operation:

Project_Rectangle($R \to \mathcal{U}', \mathcal{Q}$):
1 for all $R' \in \mathcal{U}'$ do:
2 for all $c_1 \in C_1$ and $c_2 \in C_2$ do:
3 Compute c_1', c_2', δ' with $d_{((R,c_1,c_2)\to R')}(x', y') = c_1'(x' - x_1') + c_2'(x' - x_2') + \delta'$.
4 if $\delta' < \delta_{(R',c_1',c_2')}$ then:
5 Set $\delta_{(R',c_1',c_2')} := \delta'$.
6 Set $\text{key}(R') := \min\{\text{key}(R'), \delta'\}$.
7 Set $\mathcal{Q} := \mathcal{Q} \cup \{R'\}$.

Theorem 3. *If $d_{(R,c_1,c_2)}$ for $(R, c_1, c_2) \in \mathcal{V} \times C_1 \times C_2$ are the functions produced at termination by* GeneralizedDijkstra$(G, c, \mathcal{V}, \mathcal{S})$ *implementing* Update *with* Project_Rectangle*, then $d((x, y, z)) = d_R((x, y, z))$ for all $(x, y, z) \in V(G)$. The corresponding running time of* GeneralizedDijkstra *is $O\left(k_1^2k_2^2|\mathcal{V}| \log |\mathcal{V}|\right)$.*

Proof. By Lemma 2, Project_Rectangle takes $O(k_1k_2)$ time. The number of deletions of elements from the queue is bounded by the total number of updates from any rectangle to any adjacent one, which is at most $6k_1k_2|\mathcal{V}|$. □

In our implementation, we improved the running time by applying Project_Rectangle to triples $(R, c_1, c_2) \in \mathcal{V} \times C_1 \times C_2$ instead of rectangles, where the current key is attained by $d_{(R,c_1,c_2)}$. Here, Project_Rectangle takes constant time and the number of updates from any rectangle to any adjacent one is still bounded by $O(k_1k_2|\mathcal{V}|)$. This leads to an overall running time of $O\left(k_1k_2|\mathcal{V}| \log(k_1k_2|\mathcal{V}|)\right)$.

As mentioned before, in a typical VLSI instance k_1 and k_2 can be as small as 5; see experimental results in Section 4.

3.2 Labeling Intervals

As a second application of GeneralizedDijkstra we consider the core routine in detailed routing. Its task is to find a shortest path connecting two vertex sets S and T in an induced subgraph G of G_0 with respect to costs $c : E(G) \to \mathbb{Z}_{\geq 0}$, where we can assume $c((u, v)) = c((v, u))$ for every edge $(u, v) \in E(G)$. We use the variant of GeneralizedDijkstra as described in Subsection 3.1 to compute distances $\pi(w)$ from T to each $w \in V(G)$ in a supergraph G' of G where G' is determined by the corresponding global routing corridor. Hence, $\pi(w)$ is a lower bound for the distance of each $w \in V(G)$ to T in G with respect to c, $\pi(t) = 0$ for all $t \in T$, and $\pi(u) \leq \pi(v) + c((u, v))$ for all $(u, v) \in E(G)$. We call $\pi(w)$ the

future cost of vertex w. The function π will be used to define reduced costs $c_\pi((u,v)) := c((u,v)) - \pi(u) + \pi(v) \geq 0$ for all $(u,v) \in E(G)$. We will apply GeneralizedDijkstra to find a path P from $s \in S$ to $t \in T$ in G for which $\sum_{e \in E(P)} c(e) = \pi(s) + \sum_{e \in E(P)} c_\pi(e)$ is minumum by setting $d_U(v) := \pi(v)$ for $v \in U \in \mathcal{S}$ in line 1 and using c_π instead of c.

We generalize Hetzel's algorithm [17] by using a more sophisticated future cost function π as described in the previous subsection. Note that Hetzel's algorithm strongly relies on the fact that $\pi(v)$ is the l_1-distance between v and T for all $v \in V(G)$.

As most wires use the cheap edges in preference direction, it is natural to represent the subgraph of G induced by a layer z by a set of intervals in preference direction. Horizontal intervals are rectangles of the form $[x_1, x_2] \times \{y\} \times \{z\}$, and vertical intervals are rectangles of the form $\{x\} \times [y_1, y_2] \times \{z\}$. Typically, the number of intervals is approximately 25 times smaller than the number of vertices. Hetzel [17] showed how to make Dijkstra's algorithm work on such intervals, but his algorithm only works for reduced costs defined with respect to l_1-distances. Clearly, the l_1-distance is often a poor lower bound. Therefore, our generalization allows for significant speed-ups. (In the example in Figure 3 the l_1-distance is 36, our lower bound is 130, and the actual distance is 153. The improved lower bound of 130 includes 78 units for six vias, and 16 units for the necessary detours in x- and y-direction.)

We again apply GeneralizedDijkstra. The vertex sets in $\mathcal{V}$ will be these intervals. Figure 4 illustrates the set of intervals for the example in Figure 2. In rare cases some intervals have to be split in advance to guarantee that π is monotone on an interval. A *tag* will be a triple (J, v, δ), where J is a subinterval of U, $v \in J$, and $\delta \in \mathbb{Z}_{\geq 0}$. At any stage we will have a set of tags on each interval.

A tag (J, v, δ) on $U \in \mathcal{V}$ represents the distance function $d_{(J,v,\delta)} : U \to \mathbb{Z}_{\geq 0} \cup \{\infty\}$, defined as follows: Assume that $U = [x_1, x_5] \times \{y\} \times \{z\}$ is a horizontal interval, $J = [x_2, x_4] \times \{y\} \times \{z\}$, and $v = (x_3, y, z)$ with $x_1 \leq x_2 \leq x_3 \leq x_4 \leq x_5$. Then $d_{(J,v,\delta)}((x,y,z)) := \delta + \mathrm{dist}_{(G[J],c_\pi)}(v, (x,y,z))$ for $x_2 \leq x \leq x_4$ and $d_{(J,v,\delta)}((x,y,z)) := \infty$ for $x \notin [x_2, x_4]$. Similar for vertical intervals. For $v \in U$ we define $d_U(v)$ to be the minimum of the $d_{(J,v,\delta)}(v)$ over all tags (J, v, δ) on U. However, this function d_U will not be stored explicitly.

At any stage, we have the following properties:

- If $(x,y,z), (x+1,y,z), (x+2,y,z) \in J$ for some tag (J, v, δ), then $c_\pi(((x,y,z),(x+1,y,z))) = c_\pi(((x+1,y,z),(x+2,y,z)))$ and $c_\pi(((x+2,y,z),(x+1,y,z))) = c_\pi(((x+1,y,z),(x,y,z)))$.

- The tags on an interval U are stored in a search tree and in a doubly-linked list for U, both sorted by keys, where the key of a tag (J, v, δ) is the coordinate of v which corresponds to the preference direction.

- If there are two tags $(J, v, \delta), (J', v', \delta')$ on an interval U, then
  - $v \neq v'$.
  - $J \cap J' = \emptyset$

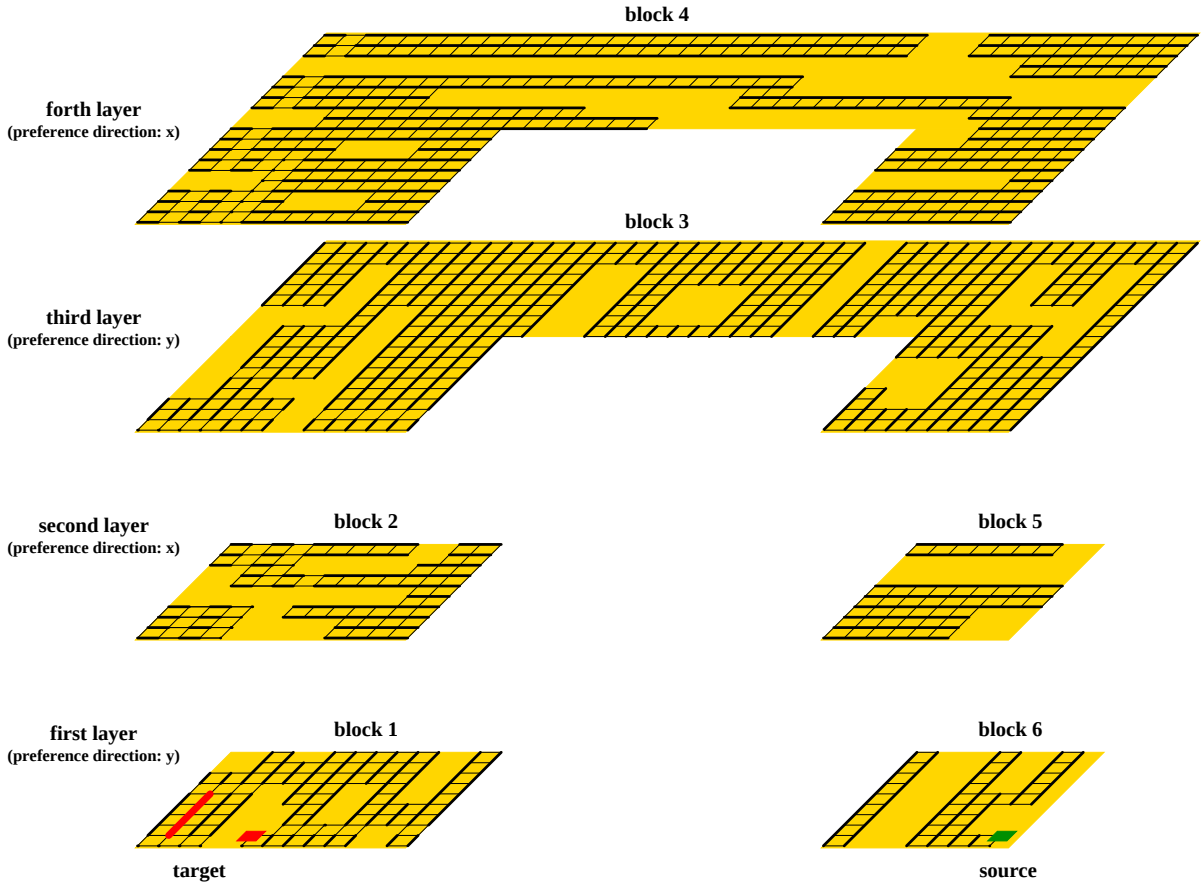

Figure 4: The intervals of the detailed routing for the instance depicted in Figure 2. For each layer, all intervals are sets of adjacent vertices according to the preference wiring direction of the layer. A block in this example is the set of intervals belonging to the same shaped area.

– $d_{(J,v,\delta)}(v') > \delta'$.

The last condition says that no redundant tags are stored. If there are k tags on U, the search tree allows us to compute $d_U(v)$ for any $v \in U$ in $O(\log k)$ time.

We can also insert another tag in $O(\log k)$ time and remove redundant tags in time proportional to the number of removed tags. As every inserted tag is removed at most once, the time for inserting tags dominates the time for removing redundant tags.

For a fixed layer z let $V_z(G')$ be the set of vertices of the supergraph G' in layer z. Then $G'[V_z(G')]$ can be decomposed into a set of maximally connected subgraphs which will become the blocks in GeneralizedDijkstra (Figure 4). Obviously, this fulfills the requirements of blocks given in Section 2. It is easy to see that the topological order of the intervals in $\mathcal{V}$ can be chosen according to non-ascending future cost values $\min\{\pi(v) \mid v \in U\}$.

The priority queue $\mathcal{Q}$ works with buckets $B_{\delta,\mathcal{W},I}$ with key $\delta \in \mathbb{Z}_{\geq 0}$, block $\mathcal{W} \in \{\mathcal{V}_1 \ldots, \mathcal{V}_N\}$ and index $I \in \{1, \ldots, |\mathcal{V}|\}$. An element $U \in \mathcal{Q} \cap \mathcal{W}$ is contained in bucket $B_{\text{key}(U),\mathcal{W},I(U)}$. The nonempty buckets are stored in a heap and processed in lexicographical order. This ensures that an interval is removed from $\mathcal{Q}$ only once per key.

The main difficulty is that an interval can have many neighbors, particularly on an adjacent layer with orthogonal wiring direction. Although a single UPDATE operation is fast, we cannot afford to perform all these operations separately. Fortunately, there is a better way.

For neighbored intervals in the same layer we need to insert at most one tag. This is due to the monotonicity of the function π on an interval. We can register labeling operations on intervals in adjacent layers in constant time by updating $\mathcal{R}$. Finally, PROJECT_REGISTERED is performed in a single step for all intervals in $\mathcal{U}$ which were registered at key λ. Set $R := \mathcal{R}(\mathcal{U}, \lambda)$ for short. We maintain a sweepline to process the elements in R which costs $O(|R| \log |R|)$ time. Hetzel [17] showed that each interval in $\mathcal{U}$ needs to be updated by at most one interval in R. Adding the time for searching neighbored intervals in R and for the labeling operation, the overall running time for PROJECT_REGISTERED applied to block $\mathcal{U}$ is $O(|R| \log |R| + |\mathcal{U}|(\log |R| + \log |\mathcal{U}| + \log \Delta_{\mathcal{U}}))$, where $\Delta_{\mathcal{U}}$ is the maximum number of tags in an interval in $\mathcal{U}$. Since $\Delta_{\mathcal{U}}$ can be bounded by the number of intervals in $\mathcal{U}$ [17], we get the following theorem:

Theorem 4. *If* GENERALIZEDDIJKSTRA *is applied to a set $\mathcal{V}$ of intervals partitioning the node set $V(G)$ of a detailed routing instance (G, c, S, T) and reduced costs c_π with respect to a feasible potential π, then its running time is*

$$O\left(\min\{(\Lambda + 1)|\mathcal{V}| \log |\mathcal{V}|, |V(G)| \log |V(G)|\}\right),$$

where Λ is the length of a shortest path from S to T with respect to c_π.

4 Experimental results

We will analyze two applications of GENERALIZEDDIJKSTRA as presented in Section 3 implemented in the detailed path search of BonnRoute which is a state-of-the-art routing tool used by IBM. Our experiments were made on an AMD-Opteron machine with 64 GB memory and four processors running at 2.6 GHz.

We have run our algorithm on 10 industrial VLSI designs from IBM. Table 1 gives an overview on our testbed which consists of seven 130 nm and three 90 nm chips of different size and of different number $|Z|$ of layers. Here, the distance between adjacent tracks of a chip is usually the minimum width of a wire plus the minimum distance of two wires. We have tested our algorithm on about 15.5 million path searches in total. We used the standard costs with which BonnRoute is applied with in practice. These are 1 and 4, resp., for edges running in and orthogonal to the preference direction, and 13 for a via.
Columns 6–9 give the sum over all instances for each chip. The number of grid nodes of all detailed routing instances sums up to 2.4 trillion. The seventh column of Table 1 shows

Chip	Tech	Chip Area in 1000 Tracks	$\|Z\|$	#Paths $\times 1000$	$\|V\|$ $\times 10^6$	$\|\mathcal{V}\|$ $\times 10^6$	$\|R\|$ $\times 10^6$	$\|R_{\text{hyb}}\|$ $\times 10^6$
Dieter	130 nm	19 × 19	6	150	21 194	862	329	11
Paul	90 nm	24 × 24	7	153	19 603	785	395	12
Lotti	130 nm	14 × 14	6	219	25 746	1 229	316	16
Hannelore	90 nm	36 × 33	7	265	40 524	1 622	904	24
Elena	130 nm	19 × 19	6	906	127 063	5 565	1 709	79
Heidi	130 nm	23 × 23	7	1 558	232 548	8 918	3 177	179
Garry	130 nm	26 × 26	7	1 834	306 343	10 820	3 893	239
Edgar	90 nm	40 × 40	7	1 893	264 988	10 538	3 850	238
Ralf	130 nm	26 × 26	7	2 941	476 217	18 616	9 852	246
Hermann	130 nm	46 × 46	7	5 582	903 919	38 577	12 435	632
All				15 501	2 418 144	97 533	36 860	1 750

Table 1: Our testbed

that the number $|\mathcal{V}|$ of intervals is by a factor of about 25 smaller than the number $|V|$ of individual vertices. This confirms observations by Hetzel [17].

In Table 2 we compare a node-based ("classical") and interval-based ("old") path search, both goal-oriented using l_1-distances as future cost (thus "classical" corresponds to Rubin's [33] algorithm and "old" to Hetzel's [17]). The interval-based implementation of Dijkstra's algorithm decreases the number of labels by a factor of about 9, while the running time is improved by a factor of 13 on average. All running times include the time for performing initialization routines for future cost queries. These numbers confirm that it is absolutely necessary to apply an interval-based path search in order to get an acceptable running time.

Chip	Number of Labels ($\times 10^6$)		Running Time (sec)	
	classical	old	classical	old
Dieter	7 257	741	9 661	607
Paul	6 144	643	8 008	580
Lotti	6 585	739	9 057	827
Hannelore	19 719	2 045	26 849	1 546
Elena	34 932	4 481	53 776	4 604
Heidi	53 314	6 529	73 156	6 160
Garry	85 194	10 571	121 189	9 715
Edgar	139 871	13 464	193 642	12 020
Ralf	82 971	10 980	123 421	11 651
Hermann	381 437	39 289	661 441	51 190
All	817 424	89 482	1 280 200	98 900

Table 2: Comparison of the node-based (classical) and interval-based (old) path search.

Second, we will compare the performance of the detailed path search in BonnRoute based on different future cost computations: Hetzel's original path search using l_1-distances for

future cost values [17], and a new version as described in Section 3.1.

The number $|R|$ of rectangles used to perform GeneralizedDijkstra on rectangles as described in Subsection 3.1 is given in the second to last column of Table 1. As this number is only by a factor of 2.6 smaller than the number of intervals, and applying GeneralizedDijkstra on rectangles can take a significant running time, it is not worthwhile to perform this pre-processing on all instances. Rather we would like to apply our new approach only to those instances where the gain in running time of the core path search routine is larger than the time spent in pre-processing. As we cannot know this a priori, we need to set up a good heuristic criterion when this effort is expected to pay off. While in one of our test scenarios ("new") we run Generalized Dijkstra on rectangles for each path search to obtain a better future cost, the second scenario ("hybrid") does this only if all of the following three conditions hold:

- the global routing corridor is the union of at least three three-dimensional cuboids.
- the number of target shapes is at most 20, and
- the l_1-distance of source and target is at least 50.

Otherwise the l_1-distance is used as future cost. We compared this to a third scenario ("old") where the l_1-distance is used always (as proposed in [17]). All scenarios run GeneralizedDijkstra on intervals as described in Section 3.2. The last column of Table 1 ($|R_{\text{hyb}}|$) lists the number of rectangles of the pre-processing step for the hybrid scenario, where we account for one rectangle if the old l_1-based approach is applied. This shows that the instance size of GeneralizedDijkstra on rectangles in the hybrid scenario is significantly smaller than that of the actual interval-based path search.

In Table 3 we compare the number of labels and the length of detours, again summed up over all instances, for each chip of our testbed. The detour of a path is given by the length of the path minus the future cost value of the source. For each chip, the sum of all path lengths, given in the last column of Table 3, was about the same for all three scenarios. (They all guarantee to find shortest paths, but they may find different shortest paths occasionally, leading to different instances subsequently.)
The number of labels clearly decreases by applying the new approach. In the hybrid and new scenario, the total number of labels can be reduced by 42% and 53%, respectively. The total length of detours decreases by 37% and 62%, respectively.

In Table 4 we present the main result of our study.
All running times for old, hybrid and new include the time for performing initialization routines for future cost queries. For the hybrid and new scenario we also give the time spent in the pre-processing routine of the new approach, which contains initializing the corresponding graph and performing GeneralizedDijkstra on rectangles. Although 4.8% of the running time of hybrid is spent in the initialization step, we obtain a total improvement of 21.7% in comparison to the old scenario. The best result of 33.3% was obtained on Hannelore, the worst of 17.2% on Elena. Only a carefully chosen heuristic enables us to identify those instances for which it is worthwhile to spend time on the more

Chip	Number of Labels ($\times 10^6$)			Length of Detours ($\times 10^3$)			Length of Paths ($\times 10^3$)
	old	hybrid	new	old	hybrid	new	
Dieter	741	469	361	5 596	4 184	2 663	36 702
Paul	643	373	273	5 688	4 363	2 838	38 544
Lotti	739	455	298	7 475	5 861	3 330	49 023
Hannelore	2 045	844	704	9 443	5 659	3 748	113 392
Elena	4 481	2 909	2 267	37 443	24 922	14 853	248 355
Heidi	6 529	4 062	2 888	58 608	40 979	22 903	404 564
Garry	10 571	6 171	4 856	77 874	48 461	30 325	605 396
Edgar	13 464	6 390	5 218	76 813	47 205	26 718	809 167
Ralf	10 980	6 657	4 771	112 238	78 915	47 280	661 519
Hermann	39 289	23 621	20 329	266 446	155 044	97 189	2 314 273
All	89 482	51 951	41 965	657 624	415 593	251 847	5 280 935

Table 3: Number of labels, length of detours and paths for the interval-based path search in three different scenarios

Chip	old	hybrid	init	new	init
Dieter	607	481	20	1 479	1 047
Paul	580	459	25	1 590	1 187
Lotti	827	688	35	1 379	815
Hannelore	1 546	1 031	48	4 355	3 370
Elena	4 604	3 811	169	8 191	4 833
Heidi	6 160	5 077	382	14 571	10 199
Garry	9 715	8 142	529	20 552	13 247
Edgar	12 020	8 627	555	19 898	12 215
Ralf	11 651	9 599	544	42 862	33 944
Hermann	51 190	39 568	1 450	83 235	43 248
All	98 900	77 483	3 757	198 112	124 105

Table 4: Running time (in sec) for the interval-based path search in three different scenarios

expensive computation of a better future cost. In the hybrid scenario the new approach is called for about 23% of the path searches, which, however, take most of the running time. We would get a theoretical improvement of 25.2% if we ran the new scenario and did not take initialization time into account. This shows that our criteria in the hybrid scenario are close to optimal.

The combination of both techniques — the interval-based path search and the usage of improved future cost values — substantially speed-up the core routine of detailed routing, one of the most time-consuming steps in the layout process. Thus, a significant reductions of the overall turn-around-time from couple of days to a few hours can be achieved.

References

[1] C. Albrecht: Global routing by new approximation algorithms for multicommodity flow. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems* **20** (2001), 622–632.

[2] H. Bast, S. Funke, D. Matijevic, P. Sanders, and D. Schultes: In Transit to Constant Time Shortest-Path Queries in Road Networks. *9th Workshop on Algorithm Engineering and Experiments (ALENEX'07)*, New Orleans, USA, SIAM (2007).

[3] S. Batterywala, N. Shenoy, W. Nicholls, and H. Zhou: Track Assignment: A Desirable Intermediate Step Between Global Routing and Detailed Routing. Proc. IEEE International Conference on Computer Aided Design, 2002, 59–66.

[4] R. Bauer and D. Delling: SHARC: Fast and Robust Unidirectional Routing. Proc. 10th International Workshop on Algorithm Engineering and Experiments (ALENEX'08), 13–26.

[5] R. Bauer, D. Delling and D. Wagner: Experimental Study on Speed-Up Techniques for Timetable Information Systems. Proc. 7th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS 2007).

[6] B.V. Cherkassky, A.V. Goldberg, and T. Radzik: Shortest Paths Algorithms: Theory and Experimental Evaluation. *Math. Prog.* **73** (1996), 129–174.

[7] J. Cong, J. Fang, and K. Khoo: DUNE — A Multilayer Gridless Routing System. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **20** (2001), 633–647.

[8] J. Cong, J. Fang, M. Xie, and Y. Zhang: MARS — A Multilevel Full-Chip Gridless Routing System. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **24** (2005), 382–394.

[9] C. Demetrescu, A.V. Goldberg, D.S. Johnson (Eds.): Shortest Path Computations: Ninth DIMACS Challenge. AMS, forthcoming.

[10] E.W. Dijkstra: A note on two problems in connexion with graphs. *Numerische Mathematik* **1** (1959), 269–271.

[11] J. Doran: An Approach to Automatic Problem-Solving. *Machine Intelligence* **1** (1967), 105–127.

[12] M.L. Fredman, and R.E. Tarjan: Fibonacci heaps and their uses in improved network optimization problems. *Journal of the ACM* **34** (1987), 596–615.

[13] G. Gallo and S. Pallottino: Shortest Paths Algorithms. *Annals of Operations Research* **13** (1988), 3–79.

[14] A.V. Goldberg: A Simple Shortest Path Algorithm with Linear Average Time. Proc. 9th European Symposium on Algorithms (ESA 2001), Lecture Notes in Computer Science LNCS 2161, Springer-Verlag, 2001, 230–241.

[15] A.V. Goldberg and C. Harrelson: Computing the Shortest Path: A* Search Meets Graph Theory. Proc. 16th ACM-SIAM Symposium on Discrete Algorithms (SODA 2005), 156–165.

[16] P.E. Hart, N.J. Nilsson, and B. Raphael: A formal basis for the heuristic determination of minimum cost paths in graphs. *IEEE Transactions on Systems Science and Cybernetics, SSC* **4** (1968), 100–107.

[17] A. Hetzel: A sequential detailed router for huge grid graphs. Proc. Design, Automation and Test in Europe (DATE 1998), 332–339.

[18] W. Heyns, W. Sansen, and H. Beke: A Line-Expansion Algorithm for the General Routing Problem with a Guaranteed Solution. Proc. 17th Design Automation Conference, 1980, 243–249.

[19] D.W. Hightower: A solution to line-routing problems on the continuous plane. Proc. 6th Design Automation Conference, 1969, 1–24.

[20] J.H. Hoel: Some Variations of Lee's Algorithm. *IEEE Transactions on Computers* **25** (1976), 19–24.

[21] M. Holzer, F. Schulz, D. Wagner, and T. Willhalm: Combining Speed-up Techniques for Shortest-Path Computations. *Journal of Experimental Algorithmics (JEA)* **10** (2005), 1–18.

[22] J. Hu and S.S Sapatnekar: A survey on multi-net global routing for integrated circuits. *Integration, the VLSI Journal* **31** (2001), 1–49.

[23] G.A. Klunder and H.N. Post: The Shortest Path Problem on Large-Scale Real-Road Networks. *Networks* **48** (2006), 182–194.

[24] E. Köhler, R.H. Möhring, and H. Schilling: Acceleration of shortest path and constrained shortest path computation. Proc. 4th International Workshop on Efficient and Experimental Algorithms (WEA 2005), Lecture Notes in Computer Science LNCS 3503, Springer-Verlag, 2005, 126–138.

[25] B. Korte, D. Rautenbach, and J. Vygen: BonnTools: Mathematical innovation for layout and timing closure of systems on a chip. *Proc. of the IEEE* **95** (2007), 555–572.

[26] U. Lauther: An Extremely Fast, Exact Algorithm for Finding Shortest Paths in Static Networks with Geographical Background. IfGIprints 22, Institut für Geoinformatik, Universität Münster (ISBN 3-936616-22-1), 2004, 219–230.

[27] C.Y. Lee: An algorithm for path connections and its application. *IRE Transactions on Electronic Computing* **10** (1961), 346–365.

[28] Y.-L. Li, H.-Y. Chen, and C.-T. Lin: NEMO: A New Implicit-Connection-Graph-Based Gridless Router With Multilayer Planes and Pseudo Tile Propagation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **26** (2007), 705–718.

[29] A. Margarino, A. Romano, A. De Gloria, Francesco Curatelli, and P. Antognetti: A Tile-Expansion Router. *IEEE Transactions on CAD of Integrated Circuits and Systems* **6** (1987), 507–517.

[30] U. Meyer: Single-Source Shortest Paths on Arbitrary Directed Graphs in Linear Average Time. Proc. 12th ACM-SIAM Symposium on Discrete Algorithms (SODA), 2001, 797–806.

[31] R.H. Möhring, H. Schilling, B. Schütz, D. Wagner, and T. Willhalm: Partitioning graphs to speed up Dijkstra's algorithm. Proc. 4th International Workshop on Efficient and Experimental Algorithms (WEA 2005), Lecture Notes in Computer Science LNCS 3503, Springer-Verlag, 2005, 189–202.

[32] I. Pohl: Bi-directional Search. *Machine Intelligence* **6** (1971), 124–140.

[33] F. Rubin: The Lee Path Connection Algorithm. *IEEE Transactions on Computers* **C-23** (1974), 907–914.

[34] P. Sanders and D. Schultes: Highway Hierarchies Hasten Exact Shortest Path Queries. Proc. 13th European Symposium Algorithms (ESA 2005), Lecture Notes in Computer Science LNCS 3669, Springer-Verlag, 2005, 568–579.

[35] F. Schulz, D. Wagner, and K. Weihe: Using Multi-Level Graphs for Timetable Information. Proc. 4th International Workshop on Algorithm Engineering and Experiments (ALENEX), Lecture Notes in Computer Science LNCS 2409, Springer-Verlag, 2002, 43–59.

[36] R. Sedgewick and J. Vitter: Shortest Paths in Euclidean Graphs. *Algorithmica* **1** (1986), 31–48.

[37] M. Thorup: Undirected Single-Source Shortest Paths with Positive Integer Weights in Linear Time. *Journal of the ACM* **46** (1999), 362–394.

[38] H.-P. Tseng and C. Sechen: A gridless multilayer router for standard cell circuits using CTM cells. *IEEE Transactions on CAD of Integrated Circuits and Systems* **18** (1999), 1462–1479.

[39] J. Vygen: Near-optimum global routing with coupling, delay bounds, and power consumption. Proc. 10th Integer Programming and Combinatorial Optimization (IPCO 2004), Lecture Notes in Computer Science LNCS 3064, Springer-Verlag, 2004, 308–324.

[40] D. Wagner and T. Willhalm: Geometric Speed-Up Techniques for Finding Shortest Paths in Large Sparse Graphs. Proc. 11th European Symposium on Algorithms (ESA 2003), Lecture Notes in Computer Science LNCS 2832, Springer-Verlag, 2003, 776–787.

[41] D. Wagner and T. Willhalm: Speed-Up Techniques Shortest Path Computations. 24th Annual Symposium on Theoretical Aspects of Computer Science (STACS 2007), Lecture Notes in Computer Science LNCS 4393, Springer-Verlag, 2007, 23–36.

[42] Z. Xing and R. Kao: Shortest Path Search Using Tiles and Piecewise Linear Cost Propagation. *IEEE Transactions on CAD of Integrated Circuits and Systems* **21** (2002), 145–158.