PathSteiner: Improving PathFinder with Quasi-Optimal Steiner-Tree Initialization

Shashwat Shrivastava*, Luka Kurešević*†, Alexandros Poupakis*, Chirag Ravishankar‡, Dinesh Gaitonde‡, Stefan Nikolić†, and Mirjana Stojilović*
*EPFL, ‡AMD, †University of Novi Sad

***Abstract*—FPGA routing has been studied for a few decades due to its critical role in determining both the quality of results and compilation time. Recent work has shown that constructing more resource-efficient routing trees can improve both runtime and wirelength. Yet, the proposed approach of constructing multiple routing trees per net via random sink orderings—within the PathFinder routing paradigm—and selecting the best candidate becomes increasingly impractical as net fanout grows. This limitation arises because the space of possible sink orderings grows super-exponentially, allowing only a small fraction of it to be explored in practice. Moreover, even exhaustive enumeration of all sink permutations for low-fanout nets does not necessarily yield an optimal routing tree. To address these limitations, we initialize routing with quasi-optimal rectilinear Steiner trees to better cover the solution space across net fanouts. Our results show that, because these trees are only weakly congestion-aware, the final routed wirelength gains are real but fall short of the wirelength advantage implied by the seeded trees. How much of the gain remains depends on how effectively PathFinder resolves congestion while staying close to the seed. We report an average 6% improvement in wirelength, along with an 8× reduction in runtime on the most congested benchmark, which otherwise takes more than half a day to route.**

I. Introduction

Field-Programmable Gate Array (FPGA) routing has been studied for decades, due to its critical role in determining the quality of results and compilation time. Traditionally, it has been formulated as a sequence of two-pin connection problems [1]. However, recent work by Shrivastava et al. [2] showed that this approach often yields suboptimal routing trees, leading to increased wirelength and longer routing runtime. These results suggest that routing is more appropriately viewed as a routing-tree optimization problem for multi-terminal nets, rather than a sequence of independent connection decisions. One approach to exploring this routing-tree space is random sink shuffling [2], which generates a forest of candidate trees by varying sink orderings and selecting the best one.

While random sink shuffling is appealing in its simplicity and effectiveness, it explores only a small subset of possible sink orderings. This limitation becomes increasingly pronounced as net fanout grows, since the number of possible sink permutations grows super-exponentially with it. As a result, sink shuffling can meaningfully explore only a limited fraction of the routing-tree solution space and influence only a small fraction of the total routed wirelength. Figure 1 quantifies this by plotting the contribution of nets in different fanout brackets to the final routed wirelength using the *Verilog-to-Routing* (VTR) router [3] with default settings. For example, even if sink permutations could be exhaustively explored for nets with ≤ 5 sinks, this would still account for $< 50\%$ of the total final wirelength.

This work is partially supported by the Swiss National Science Foundation (grant No. 182428) and the Ministry of Science, Technological Development and Innovation of the Republic of Serbia (grants No. 451-03-33/2026-03/ 200125 & 451-03-34/2026-03/ 200125).

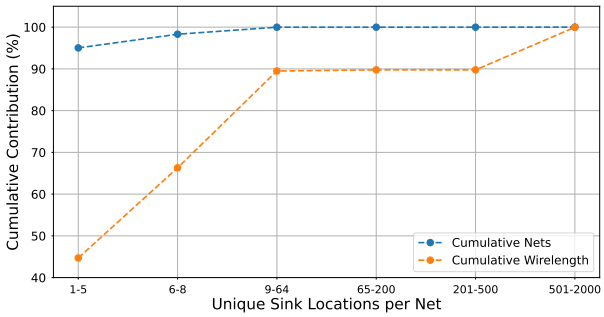

Fig. 1: Cumulative contribution of nets to wirelength as a function of the number of their unique sink locations, for the FPGA09 circuit of the *ISPD16* suite [4] with 877k nets and a Rent's exponent of 0.7.

(a) Optimal tree (b) Sink order: any

(c) Sink order: (t_1, t_2) (d) Sink order: (t_2, t_1)

Fig. 2: Illustration of the inability of exhaustive sink shuffling to guarantee route tree optimality. A router routing one sink at a time may produce any of the trees shown in figures (a), (b), or (c) if it first routes to sink t_1 and any of the trees shown in figures (a), (b), or (d) if it first routes to sink t_2.

Moreover, even exhaustive sink shuffling does not guarantee the construction of an optimal routing tree. Figure 2 provides a counterexample in which sequentially routing the sinks under any possible order may still fail to result in an optimal tree.

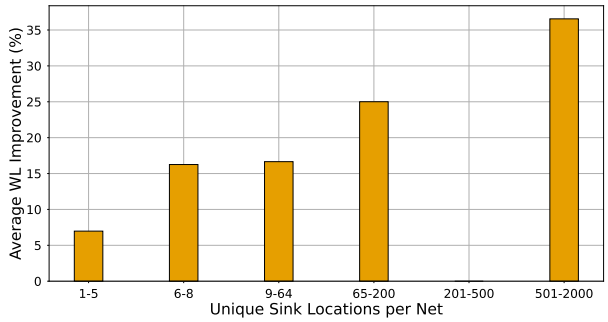

Fig. 3: Average percentage wirelength reduction achieved by FLUTE-based trees relative to the PathFinder congestion-oblivious first iteration, binned by the number of unique sink locations per net, for FPGA09 (no nets in the 201–500 bracket). For the entire circuit, average wirelength reduction is 15%.

In contrast, dedicated Steiner-tree construction algorithms such as *FLUTE* [5] can provably generate optimal *Rectilinear Steiner Minimal Trees* (RSMTs) for nets with up to nine terminals, and provide high-quality solutions within the routing-tree solution space. This capability expands the fraction of wirelength that can, in principle, be targeted by optimal tree construction, increasing it by 22% for the example in Figure 1. FLUTE's quasi-optimal tree construction for nets with higher fanout expands this even further. This can be seen in Figure 3, which shows FLUTE's average wirelength improvement per fanout bracket compared to VTR router's congestion-agnostic first iteration: clearly, FLUTE's advantage grows with fanout.

However, directly applying algorithms such as FLUTE within an FPGA router is not straightforward. FLUTE draws its efficiency from relying on lookup tables of precomputed optimal RSMTs [5], which prevents it from adapting to changing conditions and rendering it entirely congestion-oblivious. On the other hand, even querying the algorithm within an iterative loop of *PathFinder* [1] is costly and recomputing the lookup tables as congestion evolves would be prohibitive. Moreover, FLUTE operates purely on a rectilinear grid graph and does not account for FPGA-specific routing constraints such as discrete segment lengths and switch block architecture. While alternative Steiner-tree algorithms exist, they either do not scale to large nets or incur prohibitive runtime overhead [6].

This raises a natural question: can we still benefit from initializing PathFinder with congestion-oblivious quasi-optimal Steiner trees? Although starting from a congestion-oblivious solution may appear counterintuitive—because overlaps are inevitable—the first iteration of PathFinder itself is typically congestion-oblivious [1], [7], [3]. Replacing this initial step with an external quasi-optimal construction, therefore, does not discard congestion information; rather, it creates an opportunity to mitigate the inherent inefficiency in tree minimization introduced by PathFinder's default initialization.

We show that initializing PathFinder with quasi-optimal trees and carefully preserving them during routing achieves, on average, better final wirelength than sink shuffling alone. Moreover, combining our method with sink shuffling yields even further improvements. Similar to other recent works on improving routability and routed wirelength [8], [9], [10], [2], the algorithm that we propose is purely routability-driven. However, in Section X we outline how it can be equipped with timing-awareness.

II. Preliminaries

A. FLUTE

Given a set of terminals, FLUTE computes a near-optimal (for nets with > 9 terminals) or provably optimal (for nets with ≤ 9 terminals) RSMT by leveraging precomputed lookup tables [5]. Due to a favorable trade-off between optimality and efficiency, FLUTE became a widely used framework for approximating RSMTs and is a standard component in application-specific integrated circuit (ASIC) routing [11], often used as a baseline or subroutine for wirelength estimation [12] and routing optimization.

FLUTE freely places Steiner points on a plane and defines routing cost as the total Manhattan wirelength. While this is appropriate for ASIC routing, it does not account for FPGA-specific characteristics such as fixed-length routing segments or various connectivity constraints. Consequently, additional adaptation is required for it to be used in FPGA routing.

B. FPGA Architecture

The FPGA architecture model used in this work is inspired by the AMD UltraScale architectures and is similar to that of Shrivastava et al. [2]. Each routing channel comprises wires of length 1, 2, 4, and 12. We refer to these wire types as L1, L2, L4, and L12, respectively. From each *switch block* (SB) [7] and for each cardinal direction, there are 8 instances of L1, L2, and L4, and 4 instances of L12, resulting in a channel width of 208. We assume that the *Input Interconnect Block* (IIB) [10] has full connectivity and that every wire type can connect to every other type, with the following exceptions: logic block outputs and L1s cannot drive L12s, and L4s and L12s cannot drive logic block inputs. We introduce a key modification to the interconnect model of Shrivastava et al. [2] by making the SB *regular*. We do so by ensuring that each wire of a given type connects to at least one instance of every wire type it is allowed to connect to, in each direction. This modification reflects the observations of Petersen et al. about the 7-Series FPGAs [13] and reduces the impact of FLUTE's inability to take the connectivity constraints into account.

C. PathFinder

The FPGA interconnect resources can be modeled as a directed weighted graph $G = (V, E)$, called a *routing resource graph* (RRG) [1], [7], [3]. Then, given a circuit netlist and a placement solution, the routing problem amounts to finding disjoint trees embedded in G such that, for each net, its source node connects to all terminal nodes. The most successful algorithm to solve this problem is called PathFinder [1].

PathFinder is an iterative algorithm that sequentially routes nets previously decomposed into individual connections. In every iteration, it rips up non-disjoint subtrees and reroutes

them, increasing the cost of nodes as they are used. Routing finishes when the trees of all nets are disjoint, or the maximum number of iterations has elapsed.

Each RRG node u has an associated *base* cost $b(u)$, *congestion* cost $c(u)$, *capacity* $cap(u)$, and *occupancy* $occ(u)$. Occupancy is defined as the number of nets using the node. When node occupancy exceeds its capacity, the node is overused (congested), and its cost is increased to discourage its use. *Present congestion* $p(u)$ reflects the node's current occupancy, while *historical congestion* $h(u)$ reflects its overuse across routing iterations i. Equations below define these metrics.

$$c(u) = b(u) \cdot h(u) \cdot p(u) \tag{1}$$

$$p(u) = 1 + pfac \cdot \max\big(0, 1 + occ(u) - cap(u)\big) \tag{2}$$

$$h_i(u) = h_{i-1}(u) + hfac \cdot \max\big(0, occ(u) - cap(u)\big) \tag{3}$$

In the VTR [3] framework that we use in this work, each connection is routed using directed search on the RRG. The cost heuristic, called *lookahead*, estimates the cost between any node and a target pin, significantly reducing the graph exploration necessary to reach it.

III. PathSteiner

In this section, we describe the new routing flow that we call *PathSteiner*, shown in Figure 4. PathSteiner aims to initialize routing with optimal RSMTs, thereby starting with the smallest possible wirelength. However, constructing optimal Steiner trees directly on the FPGA routing graph—which is a general graph—is NP-complete [14] and impractical. We therefore reformulate the problem in terms of a rectilinear grid graph abstraction. Although minimum Steiner tree construction remains NP-hard even on grid graphs [15], methods exist to compute optimal trees for small-fanout nets and quasi-optimal trees for larger nets. We leverage one such method to generate optimized *coarse routing trees* that guide subsequent detailed routing to achieve better final routed wirelength. The router operates in three stages, each described in detail next.

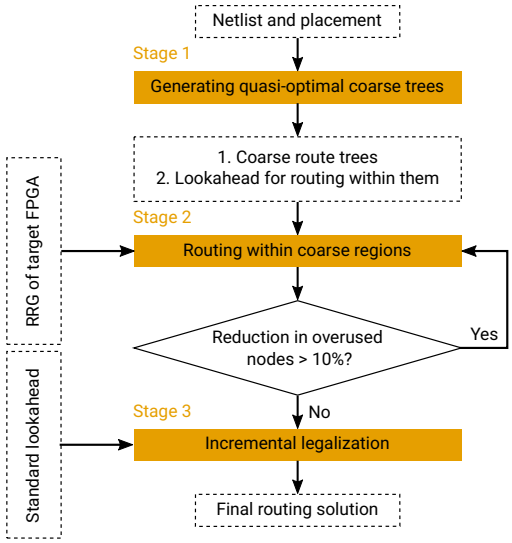

Fig. 4: PathSteiner's routing flow.

A. Generating Quasi-Optimal Rectilinear Steiner Trees

In the first stage, we construct an RSMT for each inter-tile net on a two-dimensional grid graph, where each grid point corresponds to a tile and each edge represents a routing channel between adjacent tiles. We use FLUTE because of its aforementioned computational efficiency and ability to produce optimal RSMTs for nets with ≤ 9 terminals and quasi-optimal trees for larger nets. However, the RSMT generation algorithm can be easily replaced in our flow, as shown in Section VI. Conceptually, this step takes the place of PathFinder's congestion-oblivious first iteration by initializing routing with quasi-optimal coarse trees.

FLUTE can sometimes produce diagonal edges (see Figure 5) that we decompose into equivalent L-shaped rectilinear paths [16]. This maintains the original Manhattan length, but results in a purely rectilinear coarse tree that can be implemented using the cardinal wires of modern FPGAs.

Each coarse Steiner tree output by FLUTE is represented as a set of rectilinear corridors whose endpoints are grid points corresponding to net terminals, Steiner points, or rectilinear turning points. Figure 5 illustrates these corridors in blue, with their endpoints shown in orange. Steiner points represent branching junctions in the tree, while turning points represent changes in routing direction along rectilinear corridors.

B. Routing within Coarse Regions

In the second stage, we perform detailed routing using the PathFinder-based VTR-8 router, instructing it to route each net strictly within the rectilinear corridors implied by its coarse RSMT. Routing is further constrained to prevent wire segments from overshooting Steiner points, therefore preserving intended branching points for subsequent sinks.

A coarse Steiner tree is wirelength-optimal as a whole, but individual source-sink connections along the tree may be locally suboptimal (they may not follow the shortest path), as shown in Figure 5. When using the standard VTR map lookahead, which estimates shortest-path distances between tiles, such connections can cause the router's search to devolve from a depth-first to a breadth-first traversal, significantly increasing node expansions within coarse regions. To mitigate

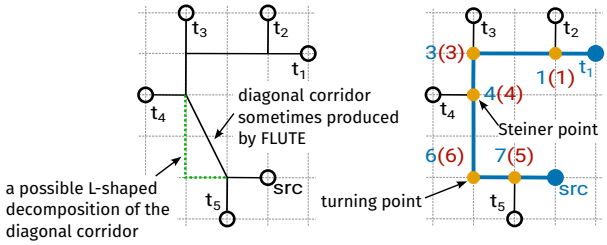

Fig. 5: Left: A FLUTE-generated optimal Steiner tree embedded in a unit-edge grid graph. Right: Rectilinear corridors used to route to sink t_1 shown in blue, with corridor endpoints (Steiner and turning points) marked in orange. For t_1 as the target, lookahead values are annotated at each endpoint: connection-specific in blue, and standard in red.

this, we introduce a new lookahead that leverages precomputed knowledge of the remaining distance along the coarse rectilinear path for each connection. For any node within a coarse region, it estimates the remaining Manhattan distance to the target based on the coarse path length, guiding the router along the intended geometry and reducing node expansions.

C. Relaxing Coarse-Region Confinement

The router may fail to resolve congestion while keeping nets strictly within the coarse trees: a phenomenon already observed before [17], [18], [2] that is worsened by the mismatch between the actual FPGA architecture and the grid graph used to generate coarse trees. To detect when the router becomes ineffective under these constraints, we monitor the reduction in the number of overused routing resource nodes between consecutive iterations: when the relative drop falls below 10%, we consider routing within coarse regions to have saturated. This threshold was determined empirically and serves as a heuristic for identifying when further iterations are unlikely to make progress. Once saturation is detected, the router transitions to the incremental legalization stage.

D. Incremental Legalization

During incremental legalization, PathFinder is allowed to route nets beyond their original coarse regions in order to resolve remaining congestion. We continue to use PathFinder's standard rip-up-and-reroute framework and switch back to the standard lookahead for the rerouted nets. Because this incremental legalization stage imposes no geometric constraints, such as the ones used by Wang et al. [8], it can produce trees that deviate significantly from the original quasi-optimal RSMTs. To counter this deviation, we introduce a simple biasing mechanism that softly encourages the router to reuse nodes within the net's original coarse region. This bias is incorporated into the node cost update equation. For a routing resource node u, the effective node cost is modified as follows:

$$c'(u) = c(u) \cdot (1 - \mathit{bias}(u)) \tag{4}$$

$$\mathit{bias}(u) = \begin{cases} \beta, & \text{if } u \in \mathcal{C}_n, \\ 0, & \text{otherwise}, \end{cases} \qquad 0 \le \beta \le 1. \tag{5}$$

$\mathcal{C}_n$ denotes the coarse region of net n, and β is a tunable bias parameter that controls the strength of the incentive.

Nodes inside the coarse region are assigned a positive bias, while those outside have zero bias. A higher bias value reduces the node cost and thus provides a stronger incentive for the router to stay inside the coarse regions obtained from the original RSMTs, whenever possible. At higher bias values, this can make the lookahead inadmissible, however.

IV. Experimental Setup

We evaluate PathSteiner on the ISPD16 benchmark suite [4] (Table I) using placements produced by the contest winner, *UTPlaceF* [19], on the architectural model described in Section II, sized at 480×168 tiles. The experiments are run on an Ubuntu 22.04 LTS machine equipped with an Intel(R) Xeon(R) CPU E5-2680 v3 @ 2.50 GHz and 252 GB of RAM.

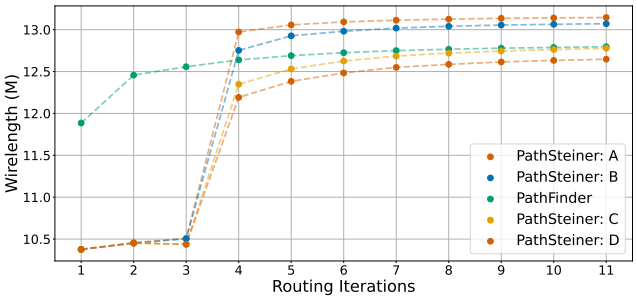

Fig. 6: Wirelength evolution over the first 11 routing iterations for FPGA05 using PathFinder and PathSteiner under progressively improved strategies. Cases A–D are defined in the column headers of Table II.

Routing is performed using the VTR8+ router [3] in non-timing-driven mode to avoid timing-driven prioritization of critical connections, enabling optimization of routing trees at the net level. To mitigate sensitivity to net ordering, we fix the present congestion cost factor (*pfac*) to five, as proposed by Zha and Li [20]. Unless otherwise stated, all remaining router parameters are set to their default VTR8+ values. One exception is the base cost assignment, which we scale according to wire-type availability. Specifically, the base cost of L1, L2, and L4 wires is set to one, while the base cost of L12 wires is set to two to reflect their lower abundance in the routing architecture. These parameter settings are used consistently across all experiments unless explicitly noted. We use FLUTE from the OpenROAD repository [21]. All artifacts required to reproduce our results are publicly available [22].

V. PathSteiner in Action

In this section, we evaluate PathSteiner to test our central hypothesis: initializing PathFinder with quasi-optimal Steiner trees leads to improved final routed wirelength. To isolate the effect of initialization alone, we disable auxiliary mechanisms such as biasing, allowing us to observe how PathFinder evolves when starting from high-quality coarse trees.

When run under this configuration, PathSteiner achieves an average 2% reduction in final routed wirelength across the benchmark suite compared to the baseline VTR router. While this improvement shows the benefit of improved initialization, it is significantly smaller than the advantage observed at the congestion-oblivious coarse-tree level (Section I). Upon closer examination of the wirelength gain per benchmark, we observe that for the most congested benchmarks—FPGA05, FPGA07, FPGA09, and FPGA11—the final wirelength is worse than the baseline. This behavior is illustrated in Figure 6 for FPGA05 (labeled "PathFinder" and "PathSteiner: A"). Although Path-

TABLE I: Characteristics of ISPD16 benchmarks [4].

FPGA	01	02	03	04	05	06	07	08	09	10	11	12
#LUTs [K]	50	100	250	250	250	350	350	500	500	350	480	500
#Nets [K]	105	168	429	430	433	713	716	725	877	961	851	1111
Rent's p	0.4	0.4	0.6	0.7	0.8	0.6	0.7	0.7	0.7	0.6	0.7	0.6

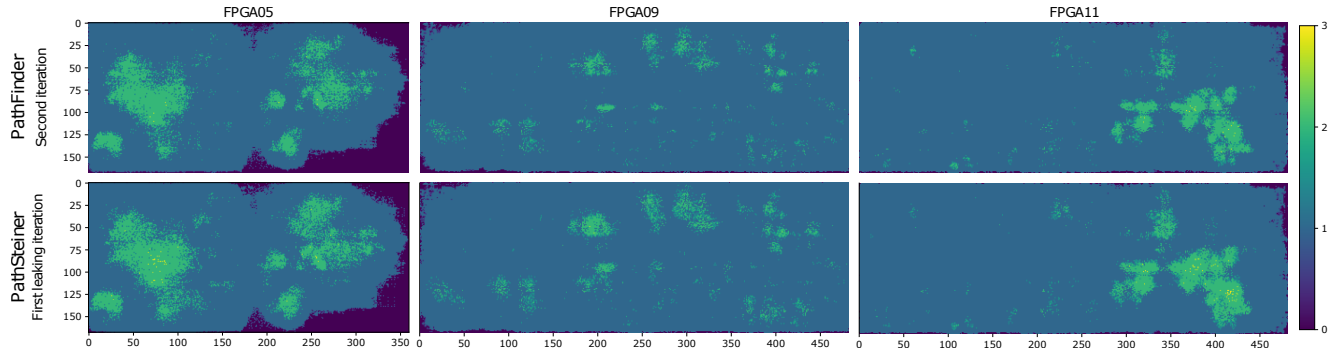

Fig. 7: Comparison of congestion maps for VTR after the second iteration and PathSteiner after the first leaking iteration (first iteration of the incremental legalization stage). Each pixel represents an FPGA tile, and its value corresponds to the maximum occupancy among the routing resources in that tile. With standard congestion resolution in coarse regions, PathSteiner exhibits larger hotspots that increase detours and worsen wirelength.

Steiner begins with a significantly better initial wirelength, the first iteration of incremental legalization (the fourth iteration for this benchmark) exhibits a sharp wirelength increase, often even surpassing the baseline. As routing converges, this deterioration persists, resulting in worse final wirelength for these benchmarks.

A. Tuning Rip-Up Strategy

The sharp increase in wirelength in the first incremental-legalization iteration suggests that, once geometric constraints are relaxed, PathFinder substantially restructures the route trees. As a first mitigation, we preserve congestion-free subtrees from the quasi-optimal coarse trees by lowering VTR's *min_incremental_fanout_threshold* (*mir*) parameter from 16 (default value) to one. This makes incremental legalization rip up only congested subtrees, rather than ripping up entire nets with fanout below 16, as VTR does by default. With $mir = 1$, FPGA07 and FPGA09 improve over the baseline by 1.9% and 0.1%, respectively, although FPGA05 and FPGA11 still deteriorate (see "PathSteiner: B" in Figure 6). Overall, the average wirelength reduction increases from 2% to 3.4% (+1.4 percentage points), indicating that partial rip-up for all nets improves solution quality. For fairness, we also apply $mir = 1$ to the baseline; it reduces wirelength by only 0.3% on average relative to $mir = 16$, consistent with VTR's default choice being driven largely by implementation efficiency [3].

B. Changing Congestion Resolution Strategy

To further investigate the deterioration in FPGA05 and FPGA11, we continue the analysis of the first iteration of incremental legalization, during which sudden, drastic wirelength deterioration occurs. We hypothesize that this deterioration arises from excessive accumulation of congestion cost. During routing within coarse regions, restricted access to routing resources forces nets to repeatedly oversubscribe the same nodes and alternate among a limited set of available resources, leading to inflated node occupancies and disproportionate growth in historical congestion cost.

To assess whether incremental legalization begins from an unusually congested state, we compare congestion maps after the first iteration of incremental legalization against those from the baseline VTR router at the end of the second iteration. They are shown in Figure 7. We select iteration two of the baseline because it is the first iteration in which PathFinder departs from the initial congestion-oblivious trees, mirroring the first incremental routing iteration in which PathSteiner shows noticeable deviation from its initial trees. We observe larger, more spatially extended congestion hotspots in PathSteiner, consistent with higher congestion and increased detours.

While this comparison does not isolate the contribution of historical cost, it motivates the hypothesis that cost accumulation during routing within coarse regions increases the wirelength observed during incremental legalization. To test this hypothesis and mitigate the observed behavior, we disable the accumulation of historical congestion cost during routing within coarse regions and rely solely on present congestion cost for negotiation between nets. Once routing within coarse regions ends, historical cost accumulation is re-enabled during incremental legalization, restoring the standard PathFinder congestion resolution mechanism. With this congestion resolution strategy, FPGA11 no longer exhibits deterioration in the final routed wirelength, and wirelength reduces by 1.5% compared to the baseline, while FPGA05 still remains worse than the baseline (see "PathSteiner: C" in Figure 6). Overall, the average routed wirelength reduction over the baseline increases from 3.4% to 4.5% (+1.1 percentage points) when using this strategy. Results for each benchmark under the aforementioned strategies are summarized in Table II.

C. Incremental Legalization Stage

Although the modified congestion resolution strategy reduces wirelength deterioration in the first legalization iteration, FPGA05 still exhibits marginal degradation at the end of routing. This suggests that fully removing geometric guidance during incremental legalization remains overly aggressive for highly congested designs. To address this, we introduce the

TABLE II: Heap operations and total wirelength of PathSteiner across routing strategies, relative to the PathFinder baseline. Heap operations are normalized. Wirelength (WL) is reported as % change (positive indicates improvement). The last row reports the geometric mean for heap operations and the arithmetic mean for wirelength.

Circuit	A: Partially rip up nets with fanout >16			B: Partially rip up nets with fanout >1			C: B + no hist. cost in Stage 2			D: C + β = 0.6		
	Pushes	Pops	Total WL	Pushes	Pops	Total WL	Pushes	Pops	Total WL	Pushes	Pops	Total WL
FPGA01	1.48×	1.38×	5.15%	1.32×	1.22×	6.09%	1.31×	1.22×	6.51%	1.34×	1.24×	7.46%
FPGA02	1.49×	1.33×	6.32%	1.35×	1.20×	6.98%	1.34×	1.20×	7.76%	1.39×	1.22×	8.37%
FPGA03	2.29×	1.49×	5.06%	2.15×	1.35×	6.39%	2.35×	1.51×	7.03%	2.39×	1.53×	7.48%
FPGA04	2.27×	1.63×	1.89%	2.11×	1.47×	4.14%	2.31×	1.64×	5.18%	2.38×	1.67×	6.05%
FPGA05	0.51×	0.39×	-2.32%	0.40×	0.28×	-1.91%	0.35×	0.22×	-0.19%	0.30×	0.16×	2.48%
FPGA06	2.44×	1.50×	3.71%	2.27×	1.35×	5.61%	2.49×	1.50×	6.25%	2.57×	1.53×	6.95%
FPGA07	2.05×	1.60×	-0.02%	1.69×	1.29×	1.91%	1.50×	1.24×	3.14%	1.62×	1.27×	4.19%
FPGA08	1.87×	1.45×	2.96%	1.73×	1.31×	5.01%	1.91×	1.47×	6.03%	1.97×	1.50×	6.63%
FPGA09	2.30×	1.64×	-1.83%	1.97×	1.43×	0.10%	1.97×	1.52×	1.56%	2.20×	1.57×	2.83%
FPGA10	1.97×	1.57×	1.59%	1.86×	1.44×	2.64%	1.99×	1.57×	3.77%	2.11×	1.60×	5.08%
FPGA11	1.57×	1.46×	-1.38%	1.26×	1.10×	-0.30%	0.77×	0.63×	1.47%	0.67×	0.52×	3.09%
FPGA12	1.56×	1.36×	2.28%	1.40×	1.22×	3.99%	1.51×	1.30×	5.11%	1.55×	1.33×	6.30%
MEAN	1.70×	1.33×	1.95%	1.51×	1.15×	3.39%	1.48×	1.13×	4.47%	**1.50**×	**1.10**×	**5.58**%

biasing mechanism that gradually relaxes adherence to the original quasi-optimal trees, as discussed in Section III.

We sweep the bias parameter $\beta \in \{0.2, 0.4, 0.6, 0.8\}$ to control how strongly routing is encouraged to stay inside the coarse regions. Even with the weakest bias, FPGA05 achieves a 1.2% wirelength reduction compared to the baseline (see "PathSteiner: D" in Figure 6). Overall, the average improvement over the baseline increases from 4.5% to 4.8% (+0.3 percentage points) with this strategy. For the best tested value, $\beta = 0.6$, this further increases to 5.6% (+0.8 percentage points).

Figure 8 shows the evolution of routing-tree similarity across routing iterations under different bias values, measured using Jaccard similarity [23]. To compute this metric, each routing tree is decomposed into unit-length rectilinear segments, preserving the coordinates of their start and end points, and the resulting segment set is compared against that of the routing tree from the first iteration (the seed). As the bias increases, Jaccard similarity increases, meaning that a larger fraction of the routed tree segments matches the initial seed and persists across iterations. This trend highlights that stronger bias increasingly constrains the search, limiting structural changes to the routing trees over time. Table II reports the wirelength reductions for each benchmark.

D. Runtime Cost of PathSteiner

Table II also reports heap push and pop counts as machine-agnostic proxies for runtime, since PathFinder's inner loop is dominated by shortest-path expansions. The time that FLUTE takes to construct the coarse trees is negligible in comparison. With the best-performing bias value ($\beta = 0.6$), PathSteiner incurs a geometric mean 1.5× increase in heap pushes relative to the baseline. This increase comes from extra routing iterations within coarse regions and from stronger biasing, which limits graph exploration and slows down convergence. The trend, however, is not uniform across all benchmarks. On FPGA05 and FPGA11, heap pushes are reduced by 3.3 and 1.5×, respectively. This suggests that PathSteiner can also improve runtime on highly congested designs. A key distinction between these two benchmarks and the rest of the suite is the structure of congestion hotspots: as shown in Figure 7, FPGA05 and FPGA11 exhibit large, spatially clustered congested regions, whereas other benchmarks (shown only for FPGA09) exhibit smaller and more dispersed hotspots.

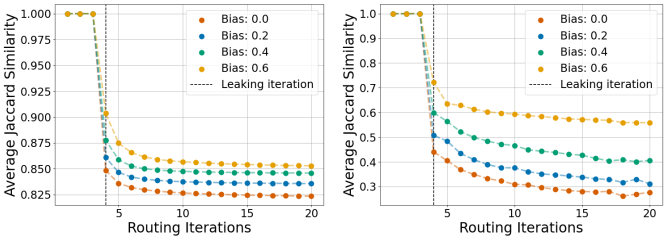

Fig. 8: Evolution of Jaccard similarity for FPGA05, under different bias values. Average similarity between route trees in the respective and the first iteration is shown (a) over all nets and (b) over nets rerouted in the respective iteration.

VI. Channel-Capacity-Aware Coarse Trees

When multiple nets select the same coarse channels, routing resources become oversubscribed, increasing the burden on PathFinder to reconstruct legal route trees in the incremental legalization stage. Given PathFinder's tendency to produce suboptimal trees [2], the likelihood of deviating from the quasi-optimal initial tree increases, forcing detours during incremental legalization and eroding the initial wirelength advantage. If a larger fraction of nets can be legalized within their coarse regions, a correspondingly larger portion of the initial quasi-optimal wirelength improvement can be preserved through convergence. The question, therefore, is how to increase legalization within coarse regions.

This motivates introducing congestion awareness during coarse tree construction. Ideally, one would like to construct rectilinear Steiner trees that adapt dynamically to congestion; however, doing so on-the-fly would preclude the use of efficient lookup-table-based methods such as FLUTE and would therefore be runtime prohibitive. As a practical alternative, we instead seek to reduce overlap in the initial coarse trees, increasing the likelihood that nets can be legalized without major restructuring during incremental legalization. Rather than developing this capability from scratch, we leverage existing

FLUTE-based ASIC global routers, which already incorporate channel-capacity awareness and provide a mechanism for generating capacity-aware rectilinear Steiner trees.

We use FastRoute4.1 [11], an open-source, channel-capacity-aware, FLUTE-based ASIC global router. It leverages FLUTE to generate initial RSMTs and subsequently refines them through capacity-aware optimization stages, including various diagonal decomposition techniques, edge shifting, and maze routing, to produce trees that better respect channel capacity constraints. To adapt FastRoute for FPGAs, we set the number of routing layers to one and disable via and layer optimization, effectively restricting routing to a 2D grid. With channel capacity matched to the target FPGA's actual channel width, FastRoute improves the average wirelength reduction from 4.5% (obtained with the capacity-unaware, unbiased RSMT initialization in Section V-B) to 4.9% (+0.4 percentage points).

A. Limits of Channel Capacity Reduction

To advance the goal of this section—improving legalization within the coarse region by becoming channel capacity aware—we explore whether artificially reducing the channel capacity for FastRoute would lead it to further avoid overlaps and enable additional reductions in the final routed wirelength. We reduce global channel capacity by 25% and 50%, while still allowing the detailed router to use all physical tracks.

For the two most congested benchmarks (FPGA05 and FPGA11), wirelength gains begin to diminish at 25% channel capacity reduction, and at 50%, FastRoute fails to generate a legal coarse solution. For FPGA09, we observed that wirelength improved marginally at 25% capacity reduction but worsened at 50% capacity. Figure 9 compares per-iteration wirelength for FPGA05 under capacity-unaware routing and capacity-aware routing at channel widths of 208 (maximum) and 156 (25% reduced). Averaged across all benchmarks, the best wirelength is achieved at the maximum channel capacity. These global capacity experiments were conducted with $\beta = 0$ in order to isolate the effect of channel capacity awareness.

Our analysis suggests two primary causes for the observed behavior: 1) reducing channel capacity leads to longer global routing trees and 2) lower capacities introduce more length-one or odd-length corridors, increasing reliance on L1 segments during detailed routing within coarse regions. Subsequently, in the incremental legalization stage, the use of longer segments (L2, L4, L12)—which do not provide intermediate switch points—increases; as a result, the router is forced into inefficient detours due to overshooting with longer segments, leading to poorer tree quality and increased wirelength.

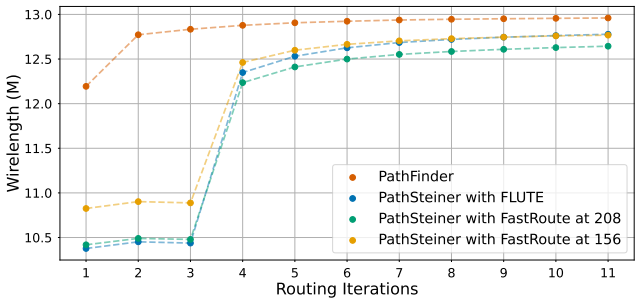

Fig. 9: Wirelength evolution in iterations 1–11 of routing FPGA05 with PathSteiner at various global channel capacities.

VII. Bringing it All Together

Combining the best-performing global channel width (among the three values evaluated) with the bias value of 0.6, identified as best in Section V-C, leads to a further 1.1 percentage point increase in average wirelength reduction, bringing the total reduction to 6% relative to the baseline. Table III reports the wirelength gains for each benchmark under this configuration, as well as the routing runtime.

Although PathSteiner increases average routing runtime by 2×, it speeds up the most congested benchmarks, FPGA05 and FPGA11, by 8.3× and 1.6×, respectively. Under PathFinder, these two benchmarks take 14 hours and one hour, compared to 0.2–12 minutes for the others, indicating that PathSteiner is particularly effective on the congested cases of primary interest in routability-driven research.

Furthermore, we compare PathSteiner with the sink shuffling approach, following the methodology of Shrivastava et al. [2]. The sink-shuffling results we obtain are slightly different because we use a more flexible routing architecture and make the VTR lookahead distance computation consistent with it. PathSteiner outperforms sink shuffling, achieving an additional 1.4 percentage point reduction in wirelength on average.

Finally, combining PathSteiner with sink shuffling yields an 8.1% reduction in wirelength relative to the baseline, as summarized in Table III. There we also list the results of combining the capacity-unaware coarse-tree generation version (using FLUTE) of PathSteiner with sink shuffling.

VIII. Related Work

Two-step routing dominated early FPGA research [24], largely due to the necessity to rely on ASIC CAD tools before FPGA-specific ones were developed [25]. Many of the early papers on FPGA routing [24], [25], [26], [18] used the *LocusRoute* ASIC global router [27], but ASIC detailed routers did not fit FPGA interconnect constraints, so dedicated alternatives such as *CGE* [25] and *SEGA* [26] emerged quickly. Despite their success, Wu and Marek-Sadowska observed that meeting global routing channel capacity poorly predicts final routability and reported superior results with a combined router [17]. Lemieux and Brown later showed that two-step routing can remain competitive if the global router is replaced by PathFinder-based *Versatile Place and Route* (VPR) [18], but their architectural assumptions do not hold for modern FPGAs. Betz then made the case for single-step PathFinder, even versus VPR+SEGA [28], [7]. From then on, interest in two-step routing mostly faded, though there were some notable exceptions [29] and global routers found use as placement congestion predictors [30], [31].

Two major differences separate classical works from ours: 1) both LocusRoute and VPR decompose nets into individual

TABLE III: Impact of channel capacity awareness and sink shuffling on PathSteiner. Heap operations (in millions) and runtime (RT, in minutes) are normalized with respect to PathFinder. Wirelength (WL) is reported as % reduction. PathSteiner is run with the best configuration: $mir = 1$, only present congestion costs while routing in coarse regions, and $\beta = 0.6$. We also report the geometric mean for heap operations and runtime, and the arithmetic mean for the final routed wirelength.

Circuit	PathFinder [3]				PathSteiner with FastRoute at Max Global Capacity				PathFinder with Sink Shuffling [2]			PathSteiner with FLUTE and Sink Shuffling			PathSteiner with FastRoute and Sink Shuffling		
	Pushes	Pops	Total WL	RT	Pushes	Pops	Total WL	RT	Pushes	Pops	Total WL	Pushes	Pops	Total WL	Pushes	Pops	Total WL
FPGA01	53	23	448,868	0.2	1.36×	1.25×	7.50%	2.15×	1.04×	1.10×	4.89%	1.41×	1.33×	8.97%	1.41×	1.33×	9.04%
FPGA02	96	42	866,076	0.3	1.51×	1.33×	8.36%	2.65×	1.05×	1.10×	5.13%	1.47×	1.31×	10.18%	1.43×	1.30×	10.37%
FPGA03	299	122	4,179,842	1.3	2.43×	1.51×	7.69%	4.11×	1.07×	1.10×	4.05%	2.54×	1.63×	8.62%	2.59×	1.60×	8.89%
FPGA04	345	127	7,138,331	1.6	2.33×	1.61×	6.39%	3.73×	1.12×	1.11×	4.53%	2.55×	1.80×	7.92%	2.51×	1.72×	8.18%
FPGA05	76,847	23,986	12,864,455	852.9	**0.16**×	**0.07**×	3.65%	**0.12**×	1.18×	1.23×	5.86%	0.36×	0.20×	6.48%	0.16×	0.07×	7.36%
FPGA06	463	185	7,706,226	2.5	2.52×	1.50×	7.29%	4.01×	1.08×	1.09×	4.01%	2.69×	1.62×	8.24%	2.69×	1.59×	8.46%
FPGA07	1,088	285	12,562,169	11.4	1.53×	1.22×	4.80%	1.83×	1.09×	1.06×	4.36%	1.80×	1.42×	6.53%	1.57×	1.28×	7.00%
FPGA08	579	215	10,473,598	2.8	1.95×	1.48×	6.84%	3.10×	1.11×	1.10×	4.62%	2.09×	1.62×	8.67%	2.08×	1.60×	8.85%
FPGA09	827	258	15,058,193	6.4	2.17×	1.52×	3.32%	2.82×	1.11×	1.09×	4.56%	2.04×	1.49×	5.91%	2.20×	1.64×	6.35%
FPGA10	507	198	9,303,661	4.0	2.00×	1.43×	5.56%	3.37×	1.05×	1.08×	2.99%	2.18×	1.70×	6.36%	2.09×	1.51×	6.71%
FPGA11	6,276	1,437	12,981,391	62.4	**0.54**×	**0.43**×	3.65%	**0.64**×	0.66×	0.63×	5.70%	0.60×	0.47×	6.67%	0.48×	0.40×	7.12%
FPGA12	589	233	7,986,825	4.9	1.53×	1.29×	6.73%	2.47×	1.06×	1.09×	4.61%	1.49×	1.30×	8.15%	1.60×	1.38×	8.49%
MEAN	716	253	8,464,136	4.45	1.38×	0.99×	5.98%	1.98×	1.04×	1.05×	4.61%	**1.55**×	**1.16**×	**7.72%**	**1.42**×	**1.04**×	**8.07%**

connections instead of building route trees holistically [25], [7], which prevents starting from trees inherently any better than those of single-step PathFinder. Under that condition, abstracting details in a two-step flow predictably hurts; 2) classical flows impose ASIC-style hard constraints on detailed routing to strictly follow the global channels, but PathFinder can fail to find detailed routing solutions even when they are guaranteed to exist [2]. We address both issues by initializing the route trees with RSMTs and treating them as soft constraints.

As designs grew, two-step routing reappeared for runtime reasons [18]. Wang et al. proposed a split global/detailed router [8], but they employed an A*-based global router that still cannot produce inherently better trees than VTR. Yet they reported wirelength gains. A direct comparison with our work is unfortunately not possible because the FPGA architecture, lookahead, and the baseline router differ. In the journal extension, Jiang et al. report significantly smaller gains [9], partly because large nets were previously omitted. They also mention coarse RSMT generation using FLUTE but dismiss it for FPGAs [9]; we show that even RSMTs that ignore channel segmentation can improve the quality of results.

Instead of viewing global route trees as soft constraints for the detailed router like we do, Wang et al. force the signals to always stay within them. However, they progressively make the congested branches of the global trees thicker. This appears to be a useful generalization of the progressive bounding-box enlargement technique of VTR [3]. Our method is much simpler to implement within any existing or future congestion-negotiation-based router, but it is possible that this approach could lead both to runtime savings and improved similarity of congestion-free trees to the initial quasi-optimal trees. It would be interesting to try to combine the two techniques.

Similarity of FPGA routing to Steiner tree construction was observed long ago [7]. Alexander et al. proposed a router that constructs Steiner trees using a heuristic blending geometric and graph formulations [32]. However, their work predates congestion negotiation [1] and was later shown to be inferior to algorithms that make use of it [18], [7]. Our approach fits modern infrastructure, though their ideas may help on-the-fly RSMT generation, which could lead to further improvements.

IX. Conclusions

In this work, we show that initializing PathFinder with quasi-optimal rectilinear Steiner trees and preserving them carefully throughout routing—by balancing congestion resolution against tree deterioration—improves the final routed wirelength. We achieve this balance via a dedicated congestion-resolution mechanism within coarse regions and a bias toward retaining quasi-optimal trees during incremental legalization. Adding channel-capacity awareness during coarse tree construction further improves results. On the most congested benchmarks, runtime reduces by up to 8×, highlighting the effectiveness of PathSteiner for challenging designs that are increasingly common today. Finally, combining with sink shuffling yields an 8% average improvement in wirelength over VTR. There are still many interesting directions in which we plan to extend the work presented here. We list a few in the next section.

X. Future work

RSMTs can elongate some critical connections and thus hurt the circuit's overall performance. This can easily be addressed by retaining the standard first iteration of PathFinder in which each net chooses a timing-optimal tree in a congestion-oblivious manner [1], [3], performing static timing analysis, and then swapping the trees of noncritical nets with their RSMTs in a step similar to area recovery in technology mapping [33]. In subsequent iterations, new nets that become timing critical can also have their RSMT swapped for a timing-optimal tree. However, since we consider RSMTs as soft constraints, the router should be able to improve timing even without that.

We showed that RSMTs unaware of FPGA routing architecture limitations can greatly improve wirelength, while enabling simple integration with rapidly evolving ASIC CAD tools. Yet, it may be possible to improve results further by offline generation of a Steiner tree database for each new FPGA architecture using an algorithm that solves the graph version of the Steiner tree minimization problem which can model the different wire types and their connectivity. The key enabler for offline generation is this paper's demonstration that initializing the router with congestion-unaware minimal trees is responsible for the bulk of the observed improvement.

REFERENCES

[1] L. McMurchie and C. Ebeling, "PathFinder: A negotiation-based performance-driven router for FPGAs," in *Proceedings of the 3rd ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. Napa Valley, CA, USA: ACM, Feb. 1995, pp. 111–17.

[2] S. Shrivastava, S. Nikolić, S. Tanaka, C. Ravishankar, D. Gaitonde, and M. Stojilović, "Guaranteed yet hard to find: Uncovering FPGA routing convergence paradox," in *Proceedings of the 2025 IEEE 33rd Annual International Symposium on Field-Programmable Custom Computing Machines*. Fayetteville, AR, USA: IEEE, May 2025, pp. 143–51.

[3] K. E. Murray, O. Petelin, S. Zhong, J. M. Wang, M. Eldafrawy, J.-P. Legault, E. Sha, A. G. Graham, J. Wu, M. J. Walker, H. Zeng, P. Patros, J. Luu, K. B. Kent, and V. Betz, "VTR 8: High-performance CAD and customizable FPGA architecture modelling," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 13, no. 2, pp. 1–60, May 2020.

[4] International Symposium on Physical Design (ISPD), "Routability-driven FPGA placement contest," https://www.ispd.cc/contests/16/ispd2016_contest.html, 2016, retrieved Aug. 2022.

[5] C. Chu and Y.-C. Wong, "FLUTE: Fast lookup table based rectilinear Steiner minimal tree algorithm for VLSI design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 1, pp. 70–83, Jan. 2008.

[6] S. E. Dreyfus and R. A. Wagner, "The Steiner problem in graphs," *Networks*, vol. 1, no. 3, pp. 195–207, Jan. 1971.

[7] V. Betz, J. Rose, and A. Marquardt, *Architecture and CAD for Deep-Submicron FPGAs*. Norwell, MA, USA: Kluwer Academic Publishers, 1999.

[8] J. Wang, J. Mai, Z. Di, and Y. Lin, "A robust FPGA router with concurrent intra-CLB rerouting," in *Proceedings of the 28th Asia and South Pacific Design Automation Conference*. Tokyo, Japan: IEEE, Jan. 2023, pp. 529–34.

[9] X. Jiang, J. Wang, J. Mai, Z. Di, and Y. Lin, "A robust FPGA router with optimization of high-fanout nets and intra-CLB connections," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 44, no. 3, pp. 1003–16, Mar. 2025.

[10] S. Shrivastava, S. Nikolić, C. Ravishankar, D. Gaitonde, and M. Stojilović, "IIBLAST: Speeding up commercial FPGA routing by decoupling and mitigating the intra-CLB bottleneck," in *Proceedings of the 42nd International Conference on Computer-Aided Design*. San Francisco, CA, USA: IEEE, Oct. 2023, pp. 1–9.

[11] M. Pan, Y. Xu, Y. Zhang, and C. Chu, "FastRoute: An efficient and high-quality global router," *VLSI Design*, vol. 2012, no. 1, pp. 1–18, Aug. 2012, art. no. 608362.

[12] J. A. Roy, J. F. Lu, and I. L. Markov, "Seeing the forest and the trees: Steiner wirelength optimization in placement," in *Proceedings of the 2006 International Symposium on Physical Design*. San Jose, CA, USA: ACM, Apr. 2006, pp. 78–85.

[13] M. B. Petersen, S. Nikolić, and M. Stojilović, "NetCracker: A peek into the routing architecture of Xilinx 7-series FPGAs," in *Proceedings of the 29th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. Virtual Event: ACM, Feb. 2021, pp. 11–22.

[14] R. M. Karp, *Reducibility among Combinatorial Problems*. Boston, MA: Springer US, 1972, pp. 85–103. [Online]. Available: https://doi.org/10.1007/978-1-4684-2001-2_9

[15] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. San Francisco, CA, USA: W. H. Freeman, 1979.

[16] M. Pan and C. Chu, "FastRoute: A step to integrate global routing into placement," in *Proceedings of the 25th International Conference on Computer-Aided Design*. San Jose, CA, USA: ACM, Nov. 2006, pp. 464–71.

[17] Y.-L. Wu and M. Marek-Sadowska, "An efficient router for 2-D field-programmable gate arrays," in *Proceedings of the European Design and Test Conference EDAC-ETC-EUROASIC*. Paris, France: IEEE, Feb. 1994, pp. 412–16.

[18] G. G. Lemieux, S. D. Brown, and D. Vranesic, "On two-step routing for FPGAs," in *Proceedings of the 1997 International Symposium on Physical Design*. Napa Valley, CA, USA: ACM, Apr. 1997, pp. 60–66.

[19] W. Li, S. Dhar, and D. Z. Pan, "Placements for ISPD16 benchmarks," http://wuxili.net/project/utplacef/, 2016, retrieved Sep. 2022.

[20] Y. Zha and J. Li, "Revisiting PathFinder routing algorithm," in *Proceedings of the 30th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. Seaside, CA, USA: ACM, Feb. 2022, pp. 24–34.

[21] The OpenROAD Project Attic, "Flute3: Open-source rectilinear Steiner minimum tree heuristic," https://github.com/The-OpenROAD-Project-Attic/flute3, 2021, archived GitHub repository; retrieved Jan. 2026.

[22] S. Shrivastava, L. Kurešević, A. Poupakis, C. Ravishankar, D. Gaitonde, S. Nikolić, and M. Stojilović, "PathSteiner—Artifacts," Available on: https://doi.org/10.5281/zenodo.19204521, 2026.

[23] P. Jaccard, "The distribution of the flora in the alpine zone," *New Phytologist*, vol. 11, no. 2, pp. 37–50, Feb. 1912.

[24] J. Rose and S. Brown, "Flexibility of interconnection structures for field-programmable gate arrays," *IEEE Journal of Solid-State Circuits*, vol. 26, no. 3, pp. 277–82, Mar. 1991.

[25] S. Brown, J. Rose, and Z. G. Vranesic, "A detailed router for field-programmable gate arrays," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 11, no. 5, pp. 620–28, May 1992.

[26] G. G. Lemieux and S. D. Brown, "A detailed routing algorithm for allocating wire segments in field-programmable gate arrays," in *Proceedings of the 4th ACM/SIGDA Physical Design Workshop*. Lake Arrowhead, CA, USA: ACM, Apr. 1993, pp. 215–26.

[27] J. Rose, "LocusRoute: A parallel global router for standard cells," in *Proceedings of the 25th Design Automation Conference*. Atlantic City, NJ, USA: ACM, Jun. 1988, pp. 189–95.

[28] V. Betz, "Architecture and CAD for speed and area optimization of FPGAs," Ph.D. dissertation, University of Toronto, Toronto, Canada, 1998. [Online]. Available: http://www.collectionscanada.ca/obj/s4/f2/dsk1/tape11/PQDD_0001/NQ41403.pdf

[29] M. Gort and J. Anderson, "Reducing FPGA router run-time through algorithm and architecture," in *Proceedings of the 21st International Conference on Field Programmable Logic and Applications*. Chania, Greece: IEEE, Sep. 2011, pp. 336–42.

[30] R. Pattison, Z. Abouwaimer, S. Areibi, G. Gréwal, and A. Vannelli, "GPlace: A congestion-aware placement tool for UltraScale FPGAs," in *Proceedings of the 35th International Conference on Computer-Aided Design*. Austin, TX, USA: IEEE/ACM, Nov. 2016, pp. 1–7.

[31] W. Li, S. Dhar, and D. Z. Pan, "UTPlaceF: A routability-driven FPGA placer with physical and congestion aware packing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 4, pp. 869–82, Apr. 2018.

[32] M. J. Alexander, J. P. Cohoon, J. L. Ganley, and G. Robins, "An architecture-independent approach to FPGA routing based on multi-weighted graphs," in *Proceedings of the European Design Automation Conference (EuroDAC94)*. Grenoble, France: IEEE Computer Society, Sep. 1994, pp. 259–64.

[33] A. Mishchenko, S. Chatterjee, and R. K. Brayton, "Improvements to technology mapping for LUT-based FPGAs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 2, pp. 240–53, Feb. 2007. [Online]. Available: https://doi.org/10.1109/TCAD.2006.887925