Provably Optimal Planar Pareto Nearest Neighbor Search with Double Monotone Chains

Zizheng Guo[1,2], Runsheng Wang[1,2,3], Yibo Lin[1,2,3,*]
[1]School of Integrated Circuits, Peking University [2]Institute of EDA, Peking University
[3]Beijing Advanced Innovation Center for Integrated Circuits
{gzz,r.wang,yibolin}@pku.edu.cn

***Abstract*—A core task in EDA is to bridge layout and topology: given planar pins, build a sparse graph that captures who should connect to whom, and then optimize on that graph. In timing-driven routing (e.g., Prim–Dijkstra), this means linking each point to its *layout* nearest neighbors. The right, metric-agnostic choice is the four-quadrant *Pareto/skyline* neighbors, which preserve candidates for any distance model—but their standard construction has a quadratic time complexity. We introduce a novel double-monotone-chain sweep algorithm that computes all Pareto neighbors in optimal, output-sensitive time $O(\sum_{i=1}^{n} k_i)$ and $O(n)$ space where n is the number of points and k_i is the number of Pareto neighbors reported for point i. This removes the $O(n^2)$ barrier while retaining full Pareto coverage. On large nets, our implementation produces Steiner trees with OpenROAD-level quality yet runs up to $39\times$ faster. The resulting primitive is a practical gateway from geometry to topology that benefits layout-aware optimizations.**

I. Introduction

Constructing sparse nearest-neighbor structures on planar point sets is a recurring bottleneck across physical design and geometric optimization. A representative example is the construction of rectilinear Steiner trees in routing and early-stage timing estimation, where a high-quality routing tree skeleton is obtained by running searches over a sparsified neighbor graph. In practice, the quality and speed of the overall flow are tightly coupled to how this graph is built.

Crucially, the "nearest neighbor" notion needed here is not tied to a fixed distance metric (e.g., L_1 or L_2). Instead, for each point p the algorithm requires the Pareto (skyline) neighbors in each of the four quadrants: those points in the quadrant that are coordinate-wise minimal (no other point in that quadrant is closer to the original point in both x and y, see Figure 1). This Pareto definition strictly supersets metric-specific neighbors and is therefore a safe and general candidate set for downstream optimization—unlike in special cases (e.g., rectilinear MST) where L_1 neighbors are proved to be suffice, many shallow-light and timing-driven problems do not admit such a guarantee.

Despite its modeling advantages, Pareto neighbor construction is under-utilized because the straightforward algorithm is $O(n^2)$: pairwise comparisons within quadrants quickly dominate runtime and memory, especially pre-DRC when nets may have degree in the hundreds of thousands. One pragmatic workaround in industry practices is to replace Pareto neighbors by eight half-octant L_1 nearest neighbors, computable in $O(n \log n)$ via divide-and-conquer; however, this option both reduces coverage (missing admissible candidates) and tends to carry a large constant, yielding slower end-to-end performance and, in our experiments, 3%–5% wirelength/path-length degradation in worst cases.

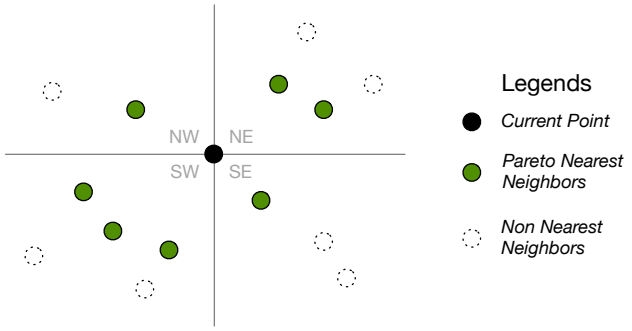

Fig. 1: Example of planar Pareto nearest neighbors.

This paper addresses the above gap. We introduce a simple yet provably optimal algorithm for planar Pareto nearest-neighbor search that matches the information-theoretic lower bound. The algorithm is output-sensitive, i.e., its running time depends on the size of the output. As the actual numbers of Pareto nearest neighbor pairs is much smaller than n^2, it brings significant runtime benefit in practice. Specifically, our resulting complexity is:

- Time: $O(\sum_{i=1}^{n} k_i)$, where k_i is the number of Pareto neighbors reported for point i (i.e., linear in the output size);
- Space: $O(n)$.

In the Prim–Dijkstra application, replacing the $O(n^2)$ skyline construction with our algorithm eliminates the dominant bottleneck and accelerates the overall flow substantially. Our main contributions are as follows.

1) *Problem framing.* We formalize planar Pareto nearest-neighbor search (quadrant skylines per point) as the

*Corresponding author.

978-3-9826741-1-7/DATE26/© 2026 EDAA

right abstraction for metric-agnostic sparse graph construction in timing-driven routing and related tasks.

2) *Output-optimal algorithm with practical simplicity.* We propose a double-monotone-chain sweep that computes all four quadrant skylines for a point set in $O(\sum_i k_i)$ time and $O(n)$ space—provably optimal up to constant factors. The method uses only sorting and array-based pointer updates, yielding low constants and easy integration.
3) *Application and evidence.* On Prim–Dijkstra shallow-light routing, our algorithm outputs Steiner trees up to 39× faster with the same wire and path length quality. Compared with the $O(n \log n)$ half-octant L_1 workaround, our approach preserves full Pareto coverage with up to 3–5% better quality while remains 30% faster thanks to better constants.

Beyond routing trees, the Planar nearest neighbor graph primitive we propose serves as a general layout-to-topology gateway that has potential use cases in many EDA fields beyond rectilinear Steiner tree construction, including buffer tree/clock tree synthesis, macro constraint-graph construction, and incorporating layout features into graph neural network models.

II. Preliminary

A. Problem Formulation: Planar Pareto Nearest Neighbors

Let $P = \{p_1, \ldots, p_n\} \subset \mathbb{R}^2$ with $p = (x_p, y_p)$. For a query anchor $q = (x_q, y_q)$ and an quadrant (quadrant) $\mathcal{O} \in \{\text{NE}, \text{NW}, \text{SE}, \text{SW}\}$, define the *quadrant range* $R_{\mathcal{O}}(q)$ in the usual way, e.g., $R_{\text{NE}}(q) = \{p \in P : x_p \geq x_q,\ y_p \geq y_q\}$. A point $p \in R_{\mathcal{O}}(q)$ is *Pareto-minimal* (a.k.a. an *quadrant skyline* element) if there is no $r \in R_{\mathcal{O}}(q)$ that is closer to q than p in both x and y directions, e.g., $x_r \leq x_p$, $y_r \leq y_p$ for the NE quadrant. The *planar Pareto nearest neighbor set* of q is

$$PN(q) = \bigcup_{\mathcal{O} \in \{\text{NE,NW,SE,SW}\}} Skyline(R_{\mathcal{O}}(q)),$$

i.e., the union of the four quadrant skylines. For the all-terminals computation considered in this paper, we set $q = p_i$ and compute $PN(p_i)$ for every $p_i \in P$. This Pareto definition is metric-agnostic (it supersets L_p-metric-specific nearest neighbor candidate sets) and, in timing-driven tree construction, avoids quality loss that can arise when restricting to L_1 or L_2 neighbors.

A straightforward baseline is to enumerate all point pairs and maintain, for each anchor q, the four quadrant skylines while sweeping in increasing x or y. Equivalently, one can scan left/right (or up/down) from each q and test dominance against all candidates. Either way, this requires $\Theta(n^2)$ pairwise comparisons in the worst case. Empirically, however, [1] reports that the *average* number of Pareto neighbors per point is much smaller than n for random point sets, and even smaller in typical placed layouts where cells align in rows.

Goal. Given P, compute $\{PN(p_i)\}_{i=1}^n$ faster than the $O(n^2)$ brute-force dominance enumeration. Ideally, the algorithm should be output-sensitive, i.e., scales with the total number of reported neighbors.

B. Nearest Neighbors & Skyline Queries

The skyline operator [2] returns undominated points under coordinate-wise dominance and underlies our quadrant skylines. Classical algorithms include BBS (branch-and-bound skyline) [3] (I/O optimal over R-trees) and presorting-based SFS [4]. These are efficient *per query* structures but, when used to compute $PN(p_i)$ for every p_i, they still induce $\Omega(n)$ queries and thus quadratic total work in the worst case. A complementary approach is the *skyline diagram* [5], which partitions the plane into cells with invariant skyline results (for quadrant, global, and dynamic skylines). While this supports $O(1)$ (plus output) query time for *arbitrary* anchors, its construction in 2D requires $\Theta(n^2)$ time/space in the worst case; hence it is unattractive for large n in EDA pre-routing stages.

C. Other Neighbor Notions in Geometry

Metric-specific neighbor graphs are extensively used in geometric optimization. For Euclidean minimum spanning tree (EMST), it suffices to consider Delaunay edges, yielding an optimal $O(n \log n)$ pipeline via Delaunay → MST [6], [7]. In rectilinear settings and special tasks, it suffices to consider L_1 nearest neighbors that can be computed using algorithms proposed by Guibas–Stolfi [8] and Zhou *et al* [9] in $O(n \log n)$ using divide-and-conquer and sweep lines, respectively. However, both L_1-only and L_2-only candidate sets can miss admissible edges in timing-driven shallow-light objectives and wider applications that are not naturally tied to a specific norm. In practice, this can introduce up to 3–5% wire and path length degradation in a PD flow.

D. Spanning/Steiner Trees for VLSI Routing

FLUTE provides near-optimal RSMT construction via lookup tables and heuristics [10], dominating wirelength-oriented flows. For balancing shallowness (path length/delay) and lightness (wirelength), shallow-light trees (SLTs) are now standard; SALT achieves strong tradeoffs with provable bounds and competitive implementations [11], [12]. Prim–Dijkstra (PD) explicitly blends MST and SPT behaviors under a tunable α [13], [1]. In addition to path length and wirelength, the direct optimization of Elmore delay has also been explored [14], [15] by taking subtree capacitance into account. Nearest neighbor search is heavily used in most of these algorithms for initializing solutions and proposing refinement candidates. This work focuses on the PD-Revisited work [1] as one example of a modern PD

pipeline: (i) neighbor graph construction, (ii) PD spanning tree with a heap and fixed α, (iii) edge flips, and (iv) detour-aware Steinerization. Among these steps, the neighbor graph construction is asymptotically dominant. We note that our neighbor search primitive is generally applicable to other lines of work as well.

Practice in open-source flows. OpenROAD integrates PD-Revisited in its pre-routing RC estimation flow [16], [17]. Notably, although [1] mentions an $O(n \log n)$ alternative based on adapting L_1 nearest-neighbor routines (e.g., Guibas–Stolfi), the *implementation* in OpenROAD defaults to the full *Pareto* (bounding-box empty) neighbor relation with an $O(n^2)$ brute-force kernel, to avoid quality loss from under-coverage by L_1-only neighbors.[1] This motivates our research on a output-sensitive Pareto nearest neighbor algorithm that enables PD/SALT-style pipelines to retain quality while removing the quadratic bottleneck.

III. Algorithm

Our method scans points sorted by x while maintaining two monotone chains along the y-order that compactly encode the current quadrant skylines on the left of the sweep line. Each new point "splices" into these chains; its Pareto neighbors in the upper-left and lower-left quadrants are enumerated by following the chains upward and downward without distance computations. Mirroring the sweep yields all four quadrants.

A. Monotone Chain

For the point set $P = \{p_1, \ldots, p_n\} \subset \mathbb{R}^2$ with $p_i = (x(i), y(i))$, we sort the point indices by x and y values. Specifically, we obtain two arrays of indices $sorted^x(\cdot)$ and $sorted^y(\cdot)$ with $x(sorted^x(si)) \leq x(sorted^x(si+1))$ and $y(sorted^y(si)) \leq y(sorted^y(si+1))$. For convenience, we also define the inverse of $sorted^x(\cdot)$ and $sorted^y(\cdot)$ as $order^x(\cdot)$ and $order^y(\cdot)$ satisfying $sorted^x(order^x(i)) = i$ for $i = 1, ..., n$ and similar for y. Throughout the paper, symbols like si and sj denote indices into sorted arrays, i.e., sweep timestamps, and symbols like i and j are indices of original points.

With that, we sweep the nodes by the order of $sorted^x(si)$. During this process, we formally define two maps of indices as follows.

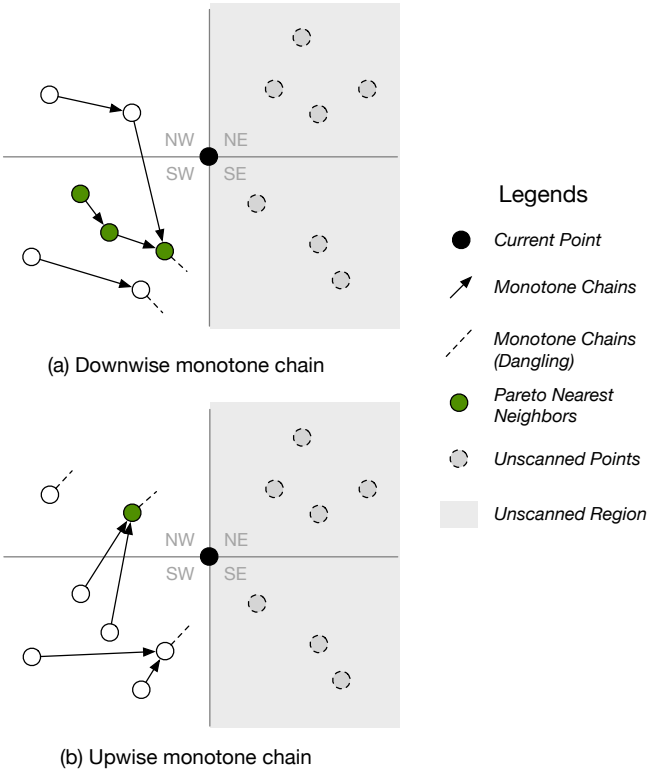

Fig. 2: Example of (a) downwise and (b) upwise monotone chains given a current sweep line status, showing its relation with the set of nearest neighbors.

$$MC_{y^-}^{(si)}(j) = \begin{cases} \text{Undefined,} & \text{if } order^x(j) \geq si, \\ k \text{ with largest } y_k \text{ satisfying} \\ \quad order^x(k) < si,\ y_k < y_j, \text{ and } x_k > x_j, \\ \text{Dangling,} & \text{if no such } k \text{ exists.} \end{cases}$$

$$MC_{y^+}^{(si)}(j) = \begin{cases} \text{Undefined,} & \text{if } order^x(j) \geq si, \\ k \text{ with smallest } y_k \text{ satisfying} \\ \quad order^x(k) < si,\ y_k > y_j, \text{ and } x_k > x_j, \\ \text{Dangling,} & \text{if no such } k \text{ exists.} \end{cases} \tag{1}$$

MC stands for *monotone chains*. These two maps of indices $MC_{y^-}^{(si)}(\cdot)$ and $MC_{y^+}^{(si)}(\cdot)$ define two directed arborescence forests that point downward and upward respectively for the current sweep position si.

Figure 2 plots the two monotone chain graphs using the same example set of points as Figure 1. Geometrically speaking, $MC_{y^-}^{(si)}(j)$ is the first point below point j that is also on the right of j in the scanned region, and $MC_{y^+}^{(si)}(j)$ is the first point above point j that is also on the right of j in the scanned region.

As we can observe, the monotone chains are directly tied to the Pareto nearest neighbors. Starting from the first Pareto

[1]See the source code comments at https://github.com/The-OpenROAD-Project/.../src/stt/src/pdr/src/pd.cpp:52.

nearest neighbor in the SW quadrant right below the current point, we only need to walk through the chain defined by $MC_{y^-}^{(si)}(\cdot)$ and collect all points we see along the way (green points in Figure 2 (a)) until the next one is Dangling. The correctness is straightforward: in order to be one of the Pareto dominants in the SW quadrant, a point k below point j has to be closer to the current point in x direction than j. That restriction gets updated once we find the first valid k below j and goes on.

A complete Pareto nearest neighbor algorithm based on monotone chains requires two missing pieces which, fortunately, can both be done efficiently: (1) locating the first two Pareto nearest neighbors in the NW and SW quadrants respectively (Section III-B), and (2) updating the monotone chain maps $MC_{y^-}^{(si)}(\cdot)$ and $MC_{y^+}^{(si)}(\cdot)$ as we sweep the points along the x direction, i.e., $si \leftarrow si + 1$ (Section III-C).

B. Locating the Start of Monotone Chain

By sorting all scanned points including the current point with increasing y values, the two points adjacent to the current point are always Pareto nearest neighbors because none of the other points can beat them in closeness to the current point in y direction. As a result, they are the start points we are looking for.

Precomputing these start points for all n points is straightforward using a separate sweep along the y direction while maintaining a classic monotone stack of points with increasing x values. This step runs in $O(n)$. We leave its complete code later in Section III-D.

C. Efficient Update of the Monotone Chain

As we sweep the points with increasing x values, we compute the Pareto nearest neighbors of current point $sorted^x(si)$ in NW and SW quadrants making use of $MC_{y^-}^{(si)}(\cdot)$ and $MC_{y^+}^{(si)}(\cdot)$. After that, we advance to the next point $sorted^x(si+1)$ and we need to update the monotone chains to $MC_{y^-}^{(si+1)}(\cdot)$ and $MC_{y^+}^{(si+1)}(\cdot)$. We have the following strong property regarding the update of monotone chains.

Theorem 1. *$MC_{y^-}^{(si)}(j) \neq MC_{y^-}^{(si+1)}(j)$ if and only if j is a Pareto nearest neighbor of $sorted^x(si)$ in the NW quadrant. Similarly, $MC_{y^+}^{(si)}(j) \neq MC_{y^+}^{(si+1)}(j)$ if and only if j is a Pareto nearest neighbor of $sorted^x(si)$ in the SW quadrant. In either case, the new value of $MC_{y^-}^{(si+1)}(j)$ or $MC_{y^+}^{(si+1)}(j)$ respectively is $sorted^x(si)$.*

Figure 3 plots the update of two monotone chains after finishing processing the last current node in Figure 2 and illustrates the above theorem. In Figure 3 (a), the downwise monotone chain of one single point is updated, which corresponds exactly to the Pareto nearest neighbor we discovered in the NW quadrant in Figure 2 (b). Similarly, Figure 3 (b)

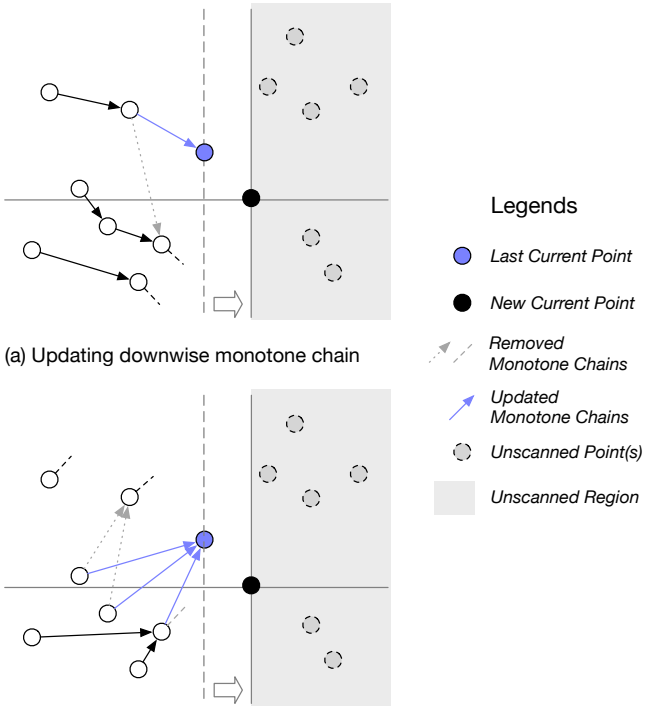

Fig. 3: Example of (a) downwise and (b) upwise monotone chain updates when a new point become scanned and get inserted to the chain.

shows 3 points updated for upwise monotone chain which corresponds exactly to the 3 green points in Figure 2 (a). The correctly updated monotone chains can then be used to compute the NW and SW nearest neighbors of the new current point as shown in Figure 4.

Although we cannot present a complete formal proof of Theorem 1 due to page limit, it is very straightforward. Intuitively, the scanned points can only "see" the newly-scanned point (i.e. point their monotone chains to the new point) if its sight is not blocked by some other points. That identifies the updated points as exactly the Pareto neighbors we just found in the last iteration. Maintaining both upwise and downwise monotone chains in a single sweep is thus an elegant implementation because these two monotone chains update each other during the sweep.

D. Full Code and Complexity Analysis

We present the complete pseudocode of our algorithm in Algorithm 1. After sorting the indices, lines 3–16 precomputes the two first Pareto neighbors in NW and SW quadrants as described in Section III-B. BC stands for *bootstrap chain* which essentially points to the two first neighbors. Generating bootstrap chains implicitly maintains two monotone stacks with increasing/decreasing x values (Figure 5). Then, lines 21–35 implements the main sweep loop. Lines 23–28

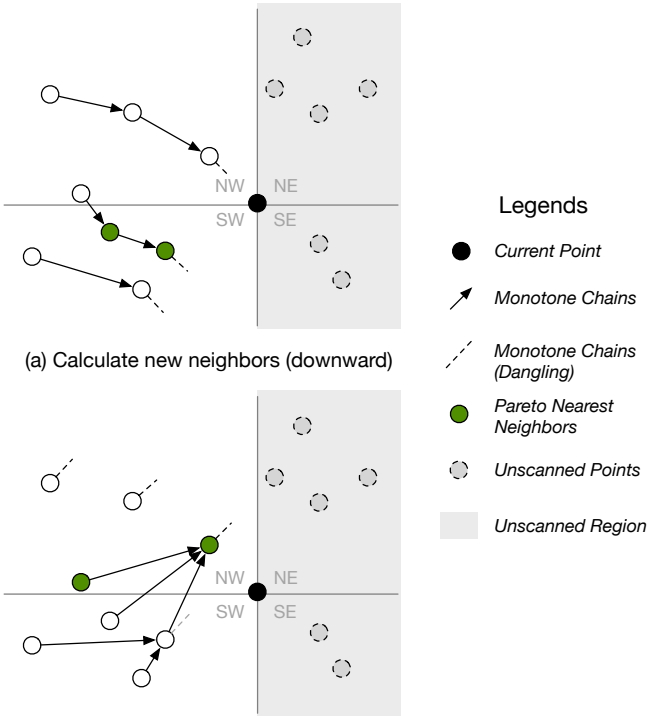

Fig. 4: Example of (a) downwise and (b) upwise updated monotone chains after Figure 3 reveals the nearest neighbor solutions for the next point.

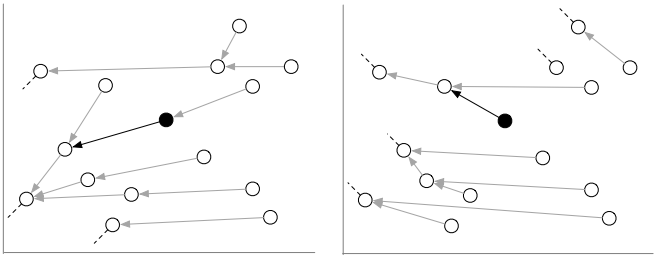

Fig. 5: Bootstrap chains provide access to the first neighbor points in NW and SW quadrants for all n points.

computes NW quadrant: starts with BC_{y^+}, traverses MC_{y^+}, and updates MC_{y^-}; lines 29–34 computes SW quadrant: starts with BC_{y^-}, traverses MC_{y^-}, and updates MC_{y^+}. The traverses and updates from two inner loops do not collide with each other because they visit points in different quadrants.

Excluding sorting, the bootstrap chain computation (lines 3–16) runs in $O(n)$ due to its analogy to monotone stacks. Each iteration of the two inner loops (lines 24–28 and 30–34) is guaranteed to insert two neighbors into the final adjacency table. As a result, the main loop (lines 21–35) runs in $O(\sum_{i=1}^{n} k_i)$, where k_i is the number of Pareto neighbors reported for point i (i.e., linear in the output size). The memory complexity is $O(n)$ because all arrays in the algorithm have length n.

Algorithm 1: The double monotone chain algorithm.

Input: $P = \{p_1, \ldots, p_n\} \subset \mathbb{R}^2$ with $p_i = (x(i), y(i))$
Output: *Neighbors*

1 $sorted^x(\cdot), sorted^y(\cdot) \leftarrow$ sorted indices of P;
2 ▷ Calculate bootstrap chains
3 $BC_{y^-} \leftarrow [\text{Dangling} \times n]$;
4 $BC_{y^+} \leftarrow [\text{Dangling} \times n]$;
5 **for** *si = 2 to n* **do**
6 $i \leftarrow sorted^y(si)$;
7 $j \leftarrow sorted^y(si-1)$;
8 **while** $j \neq$ Dangling *and* $x(j) \geq x(i)$ **do**
9 $j \leftarrow BC_{y^-}(j)$;
10 $BC_{y^-}(i) \leftarrow j$;
11 **for** *si* $= n-1$ *downto 1* **do**
12 $i \leftarrow sorted^y(si)$;
13 $j \leftarrow sorted^y(si+1)$;
14 **while** $j \neq$ Dangling *and* $x(j) \geq x(i)$ **do**
15 $j \leftarrow BC_{y^+}(j)$;
16 $BC_{y^+}(i) \leftarrow j$;
17 ▷ Calculate neighbors with monotone chains
18 $Neighbors \leftarrow [[]\times n]$;
19 $MC_{y^-} \leftarrow [\text{Undefined} \times n]$;
20 $MC_{y^+} \leftarrow [\text{Undefined} \times n]$;
21 **for** *si = 1 to n* **do**
22 $i \leftarrow sorted^x(si)$;
23 $j \leftarrow BC_{y^+}(i)$; ▷ NW quadrant;
24 **while** $j \neq$ Dangling **do**
25 Append j to $Neighbors(i)$;
26 Append i to $Neighbors(j)$;
27 $MC_{y^-}(j) \leftarrow i$;
28 $j \leftarrow MC_{y^+}(j)$;
29 $j \leftarrow BC_{y^-}(i)$; ▷ SW quadrant;
30 **while** $j \neq$ Dangling **do**
31 Append j to $Neighbors(i)$;
32 Append i to $Neighbors(j)$;
33 $MC_{y^+}(j) \leftarrow i$;
34 $j \leftarrow MC_{y^-}(j)$;
35 $MC_{y^-}(i), MC_{y^+}(i) \leftarrow$ Dangling;
36 **return** *Neighbors*

IV. Experimental Results

We implement our algorithm in under 80 lines of C++ code and integrated it into the PD-Revisited [1] flow. Our baselines include the builtin PD-Revisited implementation in the latest OpenROAD GitHub repository with $O(n^2)$ Pareto neighbors search. We do not compare with SALT

TABLE I: Runtime and quality comparisons with different nearest neighbor algorithms inside the PD-Revisited [1] flow.

Benchmark	OpenROAD Brute-Force [17]				Guibas-Stolfi [8]				Ours (Monotone NN)			
	WL/HPWL	Dist/L1	Runtime	RTR	WL/HPWL	Dist/L1	Runtime	RTR	WL/HPWL	Dist/L1	Runtime	RTR
$\alpha = 0.1$												
Group 1	**1.236**	**1.417**	3342.4	1.027	1.250	1.445	4283.4	1.316	**1.236**	**1.417**	**3254.6**	1
Group 2	**2.248**	**1.745**	2312.2	1.154	2.280	1.777	2832.8	1.414	**2.248**	**1.745**	**2003.4**	1
Group 3	**4.687**	**1.808**	1803.6	1.489	4.715	1.821	1746.6	1.442	**4.687**	**1.808**	**1211.4**	1
Group 4	**10.241**	**1.768**	2015.6	2.948	10.265	1.773	1023.0	1.496	**10.241**	**1.768**	**683.8**	1
Group 5	**22.482**	**1.719**	4450.0	9.987	22.502	1.721	636.4	1.428	**22.482**	**1.719**	**445.6**	1
Group 6	**51.809**	**1.683**	20877.6	36.208	51.823	1.686	608.2	1.055	**51.809**	**1.683**	**576.6**	1
Avg. Ratio	1	1		8.802	1.006	1.009		1.358	1	1		1
$\alpha = 0.5$												
Group 1	**1.269**	**1.291**	3379.0	1.019	1.277	1.339	4350.2	1.312	**1.269**	**1.291**	**3316.8**	1
Group 2	**2.437**	**1.423**	2352.6	1.163	2.461	1.465	2847.8	1.408	**2.437**	**1.423**	**2022.4**	1
Group 3	**5.209**	**1.461**	1798.6	1.489	5.227	1.484	1745.4	1.445	**5.209**	**1.461**	**1208.2**	1
Group 4	**11.490**	**1.466**	2000.4	2.980	11.495	1.480	1007.0	1.500	**11.490**	**1.466**	**671.2**	1
Group 5	25.308	**1.463**	4431.6	10.164	**25.290**	1.472	635.4	1.457	25.308	**1.463**	**436.0**	1
Group 6	58.406	**1.459**	20925.8	36.880	**58.343**	1.467	603.2	1.063	58.406	**1.459**	**567.4**	1
Avg. Ratio	1	1		8.949	1.003	1.017		1.364	1	1		1
$\alpha = 0.9$												
Group 1	**1.306**	**1.245**	3398.0	1.013	1.316	1.313	4407.4	1.314	**1.306**	**1.245**	**3353.0**	1
Group 2	**2.707**	**1.314**	2383.6	1.149	2.792	1.388	2875.2	1.385	**2.707**	**1.314**	**2075.4**	1
Group 3	**6.003**	**1.347**	1785.4	1.501	6.223	1.395	1728.8	1.453	**6.003**	**1.347**	**1189.6**	1
Group 4	**13.275**	**1.360**	1969.6	3.105	13.817	1.392	988.0	1.557	**13.275**	**1.360**	**634.4**	1
Group 5	**29.099**	**1.365**	4406.2	10.880	30.454	1.389	613.6	1.515	**29.099**	**1.365**	**405.0**	1
Group 6	**66.823**	**1.366**	20847.2	39.409	70.132	1.386	590.0	1.115	**66.823**	**1.366**	**529.0**	1
Avg. Ratio	1	1		9.509	1.035	1.034		1.390	1	1		1

WL/HPWL: tree wirelength normalized with net half-perimeter wirelength (HPWL).
Dist/L1: total source-sink tree path lengths normalized with total source-sink L1 distances.
Runtime: in ms. **RTR**: runtime ratio. All ratios (runtime, average WL/HPWL, average Dist/L1) assume ours as the unit 1 for easy comparison.

because PD-Revisited already outperforms SALT in runtime by $7\times$ as reported in [1]. To demonstrate the advantage of using Pareto neighbors over using half-quadrant L_1 nearest neighbors, we also implemented a special baseline adapted from the OpenROAD code by replacing Pareto neighbors search with an efficient implementation of the Guibas-Stolfi algorithm [8] in RMST-Pack [18]. We note that our Pareto neighbor search is a general primitive that can be applied to a wide range of algorithms including PD, SALT, etc.

We tested all algorithms using netlists with randomly-generated points following the practice in [9], using the statistics in Table II covering different ranges of net degrees. We run all experiments on a 64bit Linux machine with 64 cores AMD EPYC 7542 CPU and 256 GB memory. We parallelize all algorithms using OpenMP with 16 threads where the performance saturates. Every setting is run 5 times and average runtime is reported. Table I presents a detailed comparison. Our algorithm generates same-quality Steiner trees as OpenROAD while being up to $39\times$ faster on the largest nets. Compared to using only L_1 neighbors, our Pareto neighbors generate Steiner trees 35% faster and with up to 3–5% better wirelength and path length especially with PD $\alpha = 0.9$ featuring the path length objective.

TABLE II: Benchmark statistics.

Benchmark	#Nets	Degree Range	#Pins
Group 1	10^7	2–10	6×10^7
Group 2	10^6	10–50	3×10^7
Group 3	10^5	50–250	1.5×10^7
Group 4	10^4	250–1250	7.5×10^6
Group 5	10^3	1250–6250	3.75×10^6
Group 6	10^2	6250–31250	1.875×10^6

V. Conclusion

We present a novel, simple, yet provably optimal algorithm for all-pairs four-quadrant Pareto (skyline) nearest neighbors calculation. Integrated into a Steiner tree generation process, our method matches tree quality and achieves up to $39\times$ speedups on large nets. Our future work includes supporting arbitrary query points and incremental point addition, as well as exploring its broader EDA applications.

VI. Acknowledge

This work is supported in part by the National Science and Technology Major Project (2021ZD0114702), the Natural Science Foundation of Beijing, China (Grant No. Z230002), and the 111 project (B18001).

REFERENCES

[1] C. J. Alpert, W.-K. Chow, K. Han, A. B. Kahng, Z. Li, D. Liu, and S. Venkatesh, "Prim-dijkstra revisited: Achieving superior timing-driven routing trees," in *Proc. ISPD*, ser. ISPD '18. New York, NY, USA: ACM, 2018, p. 10–17.

[2] S. Börzsönyi, D. Kossmann, and K. Stocker, "The skyline operator," in *Proc. ICDE*, 2001, pp. 421–430.

[3] D. Papadias, Y. Tao, G. Fu, and B. Seeger, "An optimal and progressive algorithm for skyline queries," in *Proc. SIGMOD*, 2003, pp. 467–478.

[4] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang, "Skyline with presorting," in *Proc. ICDE*, 2003, pp. 717–719.

[5] J. Liu, J. Yang, L. Xiong, J. Pei, J. Luo, Y. Guo, S. Ma, and C. Fan, "Skyline Diagram: Efficient space partitioning for skyline queries," *IEEE Trans. Knowl. Data Eng.*, vol. 33, no. 12, pp. 3317–3331, 2021.

[6] M. I. Shamos and D. Hoey, "Closest-point problems," in *Proc. FOCS*, 1975, pp. 151–162.

[7] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars, *Computational Geometry: Algorithms and Applications*, 3rd ed. Springer, 2008.

[8] L. J. Guibas and J. Stolfi, "On computing all north-east nearest neighbors in the l1 metric," *Information Processing Letters*, vol. 17, no. 4, pp. 219–223, 1983.

[9] H. Zhou, N. S. Shenoy, and A. Sangiovanni-Vincentelli, "Efficient minimum spanning tree construction without delaunay triangulation," *Information Processing Letters*, vol. 81, no. 1, pp. 35–40, 2002.

[10] C. Chu and Y.-C. Wong, "FLUTE: Fast lookup table based rectilinear steiner minimal tree algorithm for vlsi design," *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.*, vol. 27, no. 1, pp. 70–83, 2008.

[11] G. Chen, P. Tu, and E. F. Young, "SALT: Provably good routing topology by a novel steiner shallow-light tree algorithm," in *Proc. ICCAD*, 2017, pp. 1–8.

[12] ——, "SALT: Provably good routing topology by a novel steiner shallow-light tree algorithm," *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.*, vol. 39, no. 6, pp. 1217–1230, 2020.

[13] C. J. Alpert, T. C. Hu, J.-H. Huang, A. B. Kahng, and D. Karger, "Prim-dijkstra tradeoffs for improved performance-driven routing tree design," *IEEE TCAD*, vol. 14, no. 7, pp. 890–896, 2002.

[14] H. Wu, X. Li, L. Chen, B. Yu, and W. Zhu, "Delay-driven rectilinear steiner tree construction," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2024.

[15] R. Scheifele, "Steiner trees with bounded rc-delay," *Algorithmica*, vol. 78, no. 1, pp. 86–109, 2017.

[16] T. Ajayi, V. A. Chhabria, M. Fogaça, S. Hashemi, A. Hosny, A. B. Kahng, M. Kim, J. Lee, U. Mallappa, M. Neseem, G. Pradipta, S. Reda, M. Saligane, S. S. Sapatnekar, C. Sechen, M. Shalan, W. Swartz, L. Wang, Z. Wang, M. Woo, and B. Xu, "Toward an open-source digital flow: First learnings from the openroad project," in *Proc. DAC*, ser. DAC '19. New York, NY, USA: ACM, 2019.

[17] A. B. Kahng and T. Spyrou, "The openroad project: Unleashing hardware innovation," in *Proc. GOMAC*, 2021, pp. 1–6.

[18] A. B. Khang and I. Mandoiu, "Rmst-pack: Rectilinear minimum spanning tree algorithms," http://vlsicad.ucsd.edu/GSRC/bookshelf/Slots/RSMT/RMST/, 2001.