CIRRELT

CIRRELT-2025-31

Minimal-Chain Retrieval in Binary Composition Trees for Time-Dependent Route Scheduling

Bahman Bornay
Michel Gendreau
Bernard Gendron

September 2025

Bureau de Montréal
Université de Montréal
C.P. 6128, succ. Centre-Ville
Montréal (Québec) H3C 3J7
Tél : 1-514-343-7575
Télécopie : 1-514-343-7121

Bureau de Québec
Université Laval,
2325, rue de la Terrasse
Pavillon Palasis-Prince, local 2415
Québec (Québec) G1V 0A6
Tél : 1-418-656-2073
Télécopie : 1-418-656-2624

Minimal-Chain Retrieval in Binary Composition Trees for Time-Dependent Route Scheduling

Bahman Bornay[1,2*], Michel Gendreau[1,2], Bernard Gendron[1,3]

1. Interuniversity Research Centre on Enterprise Networks Logistics and Transportation (CIRRELT)
2. Department of Mathematics and Industrial Engineering, Polytechnique Montréal
3. Department of Computer Science and Operations Research, Université de Montréal

Abstract. Route scheduling with time-dependent travel times relies on dynamic programming (DP) and repeated composition of continuous piecewise-linear (CPL) functions. During solution improvement (e.g., insertion, removal, *k*-exchange), repeatedly re-running the full DP is costly; a practical alternative is to evaluate moves *offline* by composing already cached subsequence functions. A binary composition tree supports this strategy by storing depot-to-depot function compositions; however, many queried origin - destination subsequences are not explicit tree nodes, hence one must retrieve an equivalent chain of stored nodes whose compositions reproduce the query. We show that a published retrieval rule can return chains that are *invalid* or *non-minimal*, and we provide concrete counterexamples. We then present two robust search algorithms: (i) *Canonical Decomposition Search* (with both recursive and iterative implementations) and (ii) a *Bidirectional Search*, both of which, for any queried contiguous subsequence, return a *valid* chain that is *minimal* in the number of composed functions. We formalize the alignment and containment conditions any correct chain must satisfy, characterize the specific structure of the binary composition tree used in time-dependent scheduling, and prove correctness and minimality for both algorithms. These algorithms enhance offline move evaluation: exact move costs are computed by composing cached functions, thereby avoiding route-wide rescheduling. On hundreds of routes, these methods consistently return the correct minimal chain. This note thus fixes an error in a prior tree-search variant by providing a corrected split-path retrieval and a reliable mechanism for time-dependent routing problems.

Keywords: time-dependent routing, neighborhood search, VRP, move evaluation, binary composition tree, segment tree

---

* Corresponding author: bahman.madadkar-bornay@etud.polymtl.ca

Dépôt légal – Bibliothèque et Archives nationales du Québec
Bibliothèque et Archives Canada, 2025

1 Introduction

Route scheduling in the presence of time-dependent (TD) travel times is a critical challenge in modern logistics, frequently addressed with heuristics that rely on neighborhood search. A key performance bottleneck in these heuristics is the evaluation of moves (e.g., insertions or exchanges), which often requires costly recalculations of route durations (Adamo et al. 2024). To mitigate this, offline evaluation strategies compose pre-calculated functions representing route subsequences. This composition of continuous piecewise-linear functions is a fundamental operation in solving time-dependent vehicle routing problems (TDVRPs) (Bornay, Gendreau, and Gendron 2025).

A significant advance in this area was made by Visser and Spliet (2020), who introduced a balanced binary tree to store and manage these function compositions, which they termed a Ready Time Function Tree. This data structure improves the complexity of calculating route-wide functions from $\mathcal{O}(n^2p)$ to $\mathcal{O}(np \log n)$, where n is the number of requests assigned to a route $r \in \mathscr{R}$, and p is the bound on the number of breakpoints of arrival time functions on each arc, connecting two consecutive nodes in the route sequence. Crucially, their tree-based approach allows for efficient move evaluations even when the neighborhood is searched in a non-lexicographic order, a notable advantage for implementing advanced heuristics.

For this data structure to be fully effective, a reliable method for retrieving function chains for arbitrary subsequences -those not explicitly stored as nodes- is essential. However, we have found that the specific retrieval procedure described by Visser and Spliet (2020) in their Theorem 4 is unfortunately incorrect. As we will demonstrate with concrete counterexamples based on the authors' own tree figure, their algorithm can produce composition chains that are either *invalid* (i.e., they fail to cover the requested interval) or *non-minimal*. An incorrect retrieval rule can lead to erroneous cost calculations in neighborhood search heuristics, potentially undermining the entire solution process and motivating the need for a correct and robust alternative.

In this paper, we resolve this issue by formalizing and solving the minimal-chain retrieval problem for these binary composition trees. We present two provably optimal algorithms: (i) a top-down Canonical Decomposition Search, which we adapt from the computational geometry literature where such segment trees are well-studied, and (ii) a novel, bottom-up Bidirectional Search, which we designed specifically for this problem structure. We provide formal proofs of correctness, minimality, and efficiency for both algorithms, establishing $\Theta(\log n)$ time and $\Theta(\log n)$ auxiliary space, where n is the number of leaves in the tree.

This work serves two purposes. First, it acts as a critical technical note that corrects an error in a state-of-the-art method, ensuring the research community can confidently use and build upon the Ready Time Function Tree. Second, it provides practitioners and researchers with robust, efficient, and provably optimal algorithms for a fundamental operation in time-dependent routing. Our methods can be integrated directly into offline move evaluation frameworks, making them more reliable and effective.

2 Preliminaries and Problem Formulation

To formalize the retrieval problem, we briefly review ready–time functions and the binary composition tree of Visser and Spliet (2020). We then state the validity and minimality conditions that any retrieved composition chain must satisfy.

2.1 Time-dependent Vehicle Routing Problem with Time Windows (TDVRPTW)

Consider a TDVRPTW on a directed, time-dependent customer graph $\mathscr{G} \coloneqq (\mathscr{V}, \mathscr{A})$, with $\mathscr{V} = \mathscr{V}^C \cup \{o, d\}$, where $\mathscr{V}^C$ is the set of customer locations and o, d denote the (same physical) depot. The continuous planning horizon is $\mathscr{H} \coloneqq [0, T]$ with $T \in \mathbb{R}_{\geq 0}$. Each request $m \in \mathscr{M}$ is served at some node $i \in \mathscr{V}^C$ and has a time window $[a_i, b_i] \subseteq \mathscr{H}$ with $a_i < b_i$, demand $q_m \in \mathbb{N}$, and service time $s_i \in \mathbb{R}_{\geq 0}$. For depot nodes $i \in \{o, d\}$, $q_i = 0$, $s_i = 0$, and the time window equals $\mathscr{H}$. Waiting is allowed (early arrivals may wait until a_i), but tardiness is not allowed. All arcs $(i, j) \in \mathscr{A}$ satisfy the first-in-first-out (FIFO) property (Ichoua, Gendreau, and Potvin 2003). Let $\tau_{ij}(t)$ be the continuous piecewise-linear (CPL) travel-time function on arc (i, j), and define the arrival-time function $\alpha_{ij}(t) \coloneqq t + \tau_{ij}(t)$, which is nondecreasing by FIFO. The triangle inequality need not hold. Under route-duration minimization (and/or constraints), the goal is to serve requests with a homogeneous fleet $\mathscr{K}$ of capacity Q. For each node $j \in \mathscr{V}$, define the time-window ready–time function $\theta_j(t) \coloneqq \max\{t, a_j\} + s_j$, i.e., the earliest completion time at j after arriving at time t. For a route $r = \{\dots, i, \dots, j, \dots\}$, Visser and Spliet (2020) define the ready–time function $\delta^r_{i,j}(t)$, which gives the earliest completion time at vertex j after an arrival at vertex i at time t. It is the composition

$$\delta^r_{i,j}(t) \;=\; (\theta_j \circ \alpha_{j-1,j} \circ \cdots \circ \theta_{i+1} \circ \alpha_{i,i+1} \circ \theta_i)(t), \tag{1}$$

and the optimal duration is $\Delta^*_r \;=\; \min_{t \in \mathscr{H}} (\delta^r_{o,d}(t) - t)$.

Naturally, $\delta^r_{o,d}$ can be computed in a linear fashion: start at the departure depot o, compose across the customer locations, and finish at the arrival depot d, as shown in Equation 1. Alternatively, Visser and Spliet (2020) proposed a balanced binary tree called the Ready Time Function Tree, with a bottom-up traversal of the tree: starting from the node-to-node ready–time functions $\delta^r_{i,i+1}$ computed and stored in the leaf nodes; then composing their parent nodes, e.g., $\delta^r_{i,i+2} = \delta^r_{i+1,i+2} \circ \delta^r_{i,i+1}$; and continuing this procedure until merging the two children of the root, i.e., $\delta^r_{l,d} \circ \delta^r_{o,l}$, to obtain $\delta^r_{o,d}$, where $l \in r$ is the split index at the root so that the children cover $[o, l)$ and $[l, d)$.

However, many origin–destination (OD) pairs $[i, k)$ are not represented as explicit nodes. The retrieval task is to represent such an arbitrary $[i, k)$ as a composition of already stored explicit nodes, and to do so with a *valid* and *minimal* chain. The remainder of the paper formalizes this retrieval problem and develops correct $\Theta(\log n)$-time retrieval algorithms (where n is the number of leaves in the tree).

2.2 The Binary Composition Tree

To accelerate the repeated computation of $\delta^r_{i,j}$, Visser and Spliet (2020) proposed storing precomputed compositions in a balanced binary tree, $\mathscr{T}^r$. We adopt this structure and formalize its properties via the lemmas below.

Route sequence and interval convention. Let $\sigma(r)$denote the visit sequence of route r. We index positions along $\sigma(r)$ by $\{0, 1, \ldots, n\}$. We use *half-open* intervals $[\,\cdot, \cdot\,)$: the leaf at position t corresponds to $[\,t, t+1\,)$ and stores $\delta^r_{t,t+1}$. A tree node $\delta^r_{a,b}$ stores the composition over $[\,a, b\,)$. A query $[\,i, k\,)$ asks for the unique minimal chain of explicit nodes whose left-to-right union is exactly $[\,i, k\,)$ (adjacency means consecutive endpoints match: $b_t = a_{t+1}$), and the evaluation order is right-to-left.

Lemma 1 (Fullness under left-to-right pairing) *Consider the bottom-up construction that, at each level ℓ, pairs nodes left-to-right $((v_0, v_1), (v_2, v_3), \ldots)$ to create parents at level $\ell{+}1$ and, if the count is odd, carries the rightmost node unpaired to level $\ell{+}1$. Then every internal node in the resulting binary composition tree has exactly two children (the tree is full).*

Proof. At each level, every parent is created from a *pair* of children, hence has two children. An unpaired node at level ℓ is *not* made a parent at level ℓ; it is merely promoted to level $\ell{+}1$, where it is eventually paired to make a parent with two children. No step creates a one-child internal node. □

Lemma 2 (Left-leaning imbalance pattern) *Under the same construction, whenever an internal node has one leaf child and one non-leaf child, the leaf child is the* right *child and the non-leaf child is the* left *child.*

Proof. If a level has an odd number of nodes, the unpaired node is the *rightmost* block at that level. When it is finally paired at the next level, its sibling lies to its left and has (weakly) larger span, hence is non-leaf when the unpaired block is a leaf. Therefore the parent has a non-leaf left child and a leaf right child. The reverse configuration would require the unpaired block to sit on the left boundary, which cannot occur under left-to-right pairing. □

Proposition 3 (Structure of the Binary Composition Tree) *The binary tree, such as those illustrated in this work, exhibits the following structural properties:*

1. Leaf-consecutive-node Correspondence: *The leaf nodes of the tree correspond to computations between consecutive nodes in the route sequence.*
2. Left-leaning Imbalance Pattern: *The tree is not necessarily perfectly balanced. When imbalance occurs at an internal node, it may appear as a non-leaf left child and a leaf right child. However, the reverse, a leaf left child and a non-leaf right child, does not occur.*
3. Bounded Leaf Depth: *The height difference between any two leaf nodes in the tree is at most 1.*

4. Completeness vs. Fullness: *The tree need not be complete, but it is full.*

Proof. The properties follow from the bottom-up construction of the tree over a sequence of n leaves.

1. **(Leaf-consecutive-node Correspondence)** Proof by construction. The tree is built from the bottom up. The lowest level of the tree, the leaves, represents the most granular compositions possible. These are the functions that span the elementary intervals between consecutive nodes i and $i+1$ in the route. All higher-level internal nodes are formed by composing these foundational leaf nodes.

2. **(Left-leaning Imbalance)** Proof by construction. At each level of construction, nodes are paired from left to right to form their parents in the level above. If a level contains an odd number of nodes, the rightmost node is left without a partner and is paired one level higher, adjacent to a more developed subtree on its left. Hence an internal node may have a non-leaf left child and a leaf right child, but not the reverse (Lemma 2).

3. **(Bounded Leaf Depth)** The construction builds the tree upwards level by level. If n is not a power of two, some nodes are promoted one level earlier than others (as in item 2), but the systematic pairing ensures all leaves lie on one of two adjacent levels. Thus, the maximum depth difference between any two leaves is at most 1.

4. **(Completeness vs. Fullness)** Non-completeness: the lowest level need not be filled strictly left-to-right (e.g., a route sequence $\sigma(r) = \{0, 1, \ldots, 13\}$ yields 13 leaves and an imbalanced last level). Fullness: by Lemma 1, every internal node is created from a pair of children, hence has exactly two children. The left-leaning pattern of Lemma 2 is fully consistent with fullness.□

2.3 Conditions for Correct Chain Retrieval

For a query $\delta^r_{i,k}$ that does not correspond to an explicit node in $\mathscr{T}^r$, a chain of existing nodes must be retrieved. Any such chain must satisfy fundamental conditions of validity and optimality.

Lemma 4 (Composition Alignment) *A composition $\delta^r_{j,k} \circ \delta^r_{i,j}$ is valid if and only if the second index of the inner function equals the first index of the outer function. The result is the extended function $\delta^r_{i,k}$.*

Proof. By definition, $\delta^r_{a,b}$ maps a departure at node a to a departure at b along the contiguous subsequence $[\, a, b\,)$. A composition $\delta^r_{j,k} \circ \delta^r_{i,j}$ is well-typed if and only if the codomain index of the inner function equals the domain index of the outer map, i.e., j. The composed map follows the concatenated subsequence from i to k, yielding $\delta^r_{i,k}$. □

Lemma 5 (Boundary Alignment and Interval Containment) *Any valid composition chain for a query $\delta^r_{i,k}$ must (i) start with a function whose first index equals i and end with a function whose second index equals k; (ii) satisfy the adjacency condition of Lemma 4 at every junction; and (iii) use only indices contained in $[\, i, k\,)$.*

Proof. (Only if.) A chain representing $\delta^r_{i,k}$ must start at i and end at k; otherwise, its domain or codomain would be incorrect. Adjacency at every junction is required for the composition to be well-typed. If any intermediate $\delta^r_{a,b}$ uses an index outside $[\,i, k\,)$, the chain would traverse a noncontiguous interval, which cannot produce the map on $[\,i, k\,)$. (If.) Conversely, any chain satisfying these conditions is a composition over a valid partition of $[\,i, k\,)$ and thus equals $\delta^r_{i,k}$. □

Definition 6 (Minimal Composition Chain) *Among all valid chains for $\delta^r_{i,k}$ that satisfy Lemmas 4–5, a chain is* minimal *if it uses the fewest functions from the tree.*

With these preliminaries established, we can now formally analyze the algorithm from prior work.

3 Analysis of the Prior Tree Search Algorithm

In this section we formalize the retrieval rule described by Visser and Spliet (2020) and make explicit the ambiguities that affect correctness and minimality. We then specify two literal interpretations of the rule to be evaluated by counterexample in Section 3.4.

3.1 Statement Under Analysis (verbatim)

Theorem 4, pages 10 and 11 of Visser and Spliet (2020):

> "The most efficient nodes in the tree for the composition can now be found to be all *right* child nodes of the nodes along the (left) search path of $\delta^r_{i-1,i}$ and all the *left* child nodes of the nodes along the (right) search path of $\delta^r_{j,j+1}$."

The intended context is the tree $\mathscr{T}^r$, the two leaves adjacent to the query interval, $\delta^r_{i-1,i}$ and $\delta^r_{j,j+1}$, and their lowest common ancestor (the *split* node). We show below that, taken literally, the quoted rule is underspecified and may return chains that are either *invalid* (contain indices outside $[\,i, j\,)$) or *non-minimal* (contain more nodes than necessary).

3.2 Formal Preliminaries for Search Paths

Let $\mathrm{LCA}(u, v)$ denote the lowest common ancestor (LCA) of nodes u and v in $\mathscr{T}^r$, and let

$$\mathrm{split}(i, j) \;:=\; \mathrm{LCA}\big(\delta^r_{i-1,i},\, \delta^r_{j,j+1}\big).$$

Define the *left search path* $P_L(i, j)$ to be the unique path of nodes from $\delta^r_{i-1,i}$ up to (and including) $\mathrm{split}(i, j)$, and the *right search path* $P_R(i, j)$ the unique path from $\delta^r_{j,j+1}$ up to (and including) $\mathrm{split}(i, j)$. For a non-leaf node x, let $\textsc{Left}(x)$ and $\textsc{Right}(x)$ denote its left and right children, respectively; let $\textsc{Sibling}(x)$ denote the other child of $\textsc{Parent}(x)$. For any node $\delta^r_{a,b}$, write $\mathrm{interval}(\delta^r_{a,b}) = [\,a, b\,)$.

3.3 Two Literal Interpretations of the Quoted Rule

The wording "all right child nodes of the nodes along the (left) search path" and "all left child nodes of the nodes along the (right) search path" admits (at least) two concrete readings. We formalize both, intentionally *without* adding containment or stopping conditions that are not stated in the quote.

(L1) Downward–Children interpretation. Starting at $x = \mathrm{split}(i, j)$, traverse *downwards* toward the leaves *along* the two search paths. Define the two sets:

$$\mathcal{C}_L^{\downarrow}(i,j) := \{\textsc{Right}(x) \ : \ x \text{ lies on the leftward descent from } \mathrm{split}(i,j) \text{ to } \delta^r_{i-1,i}\},$$

$$\mathcal{C}_R^{\downarrow}(i,j) := \{\textsc{Left}(x) \ : \ x \text{ lies on the rightward descent from } \mathrm{split}(i,j) \text{ to } \delta^r_{j,j+1}\}.$$

The returned chain is any ordering of $\mathcal{C}_L^{\downarrow}(i,j) \cup \mathcal{C}_R^{\downarrow}(i,j)$ that obeys the adjacency condition of Lemma 4. No further filtering (e.g., containment in $[\,i, j\,)$) is specified by (L1).

(L2) Upward–Children interpretation. Starting at the two leaves, *ascend* to the split. Each time the ascent on the left path crosses an edge from a left child to its parent, record the parent's *right child*; each time the ascent on the right path crosses an edge from a right child to its parent, record the parent's *left child*. Formally,

$$\mathcal{C}_L^{\uparrow}(i,j) := \{\textsc{Right}(\textsc{Parent}(x)) \ : \ x \in P_L(i,j) \text{ and } x = \textsc{Left}(\textsc{Parent}(x))\},$$

$$\mathcal{C}_R^{\uparrow}(i,j) := \{\textsc{Left}(\textsc{Parent}(x)) \ : \ x \in P_R(i,j) \text{ and } x = \textsc{Right}(\textsc{Parent}(x))\}.$$

Again, the quoted rule specifies neither a containment test nor a stop condition at the split node; hence, (L2) returns any ordering of $\mathcal{C}_L^{\uparrow}(i,j) \cup \mathcal{C}_R^{\uparrow}(i,j)$ that respects the adjacency condition of Lemma 4.

Remark 7 (Missing qualifications in the quoted rule) *Neither (L1) nor (L2) enforces (i) containment* $\mathrm{interval}(\cdot) \subseteq [\,i, j\,)$*, or (ii) the standard segment-tree stop at the split (with boundary exception). As shown next, omitting these qualifications leads to invalid or non-minimal chains.*

Paper example $[\,1, 15\,)$ and the two readings. For the tree of Figure 1, (L1) (downward) returns the set $\{\delta^r_{1,2}, \delta^r_{2,4}, \delta^r_{4,8}, \delta^r_{8,12}, \delta^r_{12,14}, \delta^r_{14,15}\}$, which composes in the usual evaluation order (right-to-left) as

$$\delta^r_{14,15} \circ \delta^r_{12,14} \circ \delta^r_{8,12} \circ \delta^r_{4,8} \circ \delta^r_{2,4} \circ \delta^r_{1,2}.$$

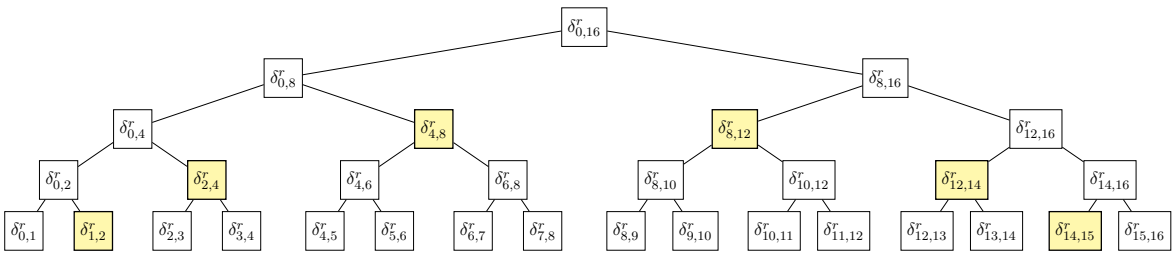

Figure 1: Binary composition tree for query $\delta^r_{1,15}$. Under the Downward–Children interpretation (L1), the highlighted nodes (yellow) form a valid minimal chain that covers the query interval $[\,1, 15\,)$.

In contrast, (L2) (upward) *without* an explicit "stop at split" adds the split's opposite child $\delta^r_{0,8}$, thereby violating Lemma 5(iii). Hence, unless one adds a stop condition not present in the quoted text, (L2) yields an invalid chain even on this canonical example.

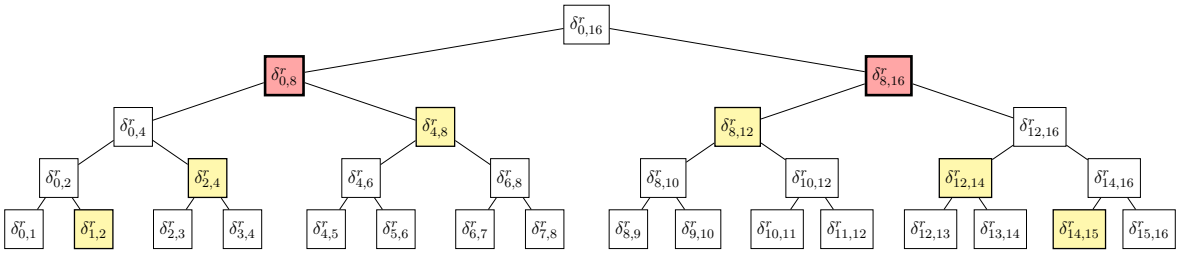

Figure 2: Upward interpretation (L2) without a split stop: both children of the split node, $\delta^r_{0,8}$ and $\delta^r_{8,16}$ (red), are incorrectly added; correct nodes are yellow. Query $\delta^r_{1,15}$ over $[\,1, 15\,)$.

3.4 Counterexample Framework

We use the canonical tree with leaves $\{\delta^r_{0,1}, \delta^r_{1,2}, \ldots, \delta^r_{15,16}\}$ and internal nodes as in Figure 1 (root $\delta^r_{0,16}$ with left child $\delta^r_{0,8}$ and right child $\delta^r_{8,16}$). For each query we list the nodes returned by (L1) or (L2), then the correct minimal chain (Definition 6).

Counterexample A (crossing the root split on a non-power-of-two tree tree).

Tree with root $\delta^r_{0,23}$ and children $\delta^r_{0,12}$ and $\delta^r_{12,23}$. Query $[\,i, j\,) = [\,10, 18\,)$ crosses the root split at 12.

Minimal chain.

$$\{\delta^r_{10,12},\ \delta^r_{12,18}\} \xrightarrow{\text{leading to}} \delta^r_{10,18} = \delta^r_{12,18} \circ \delta^r_{10,12}.$$

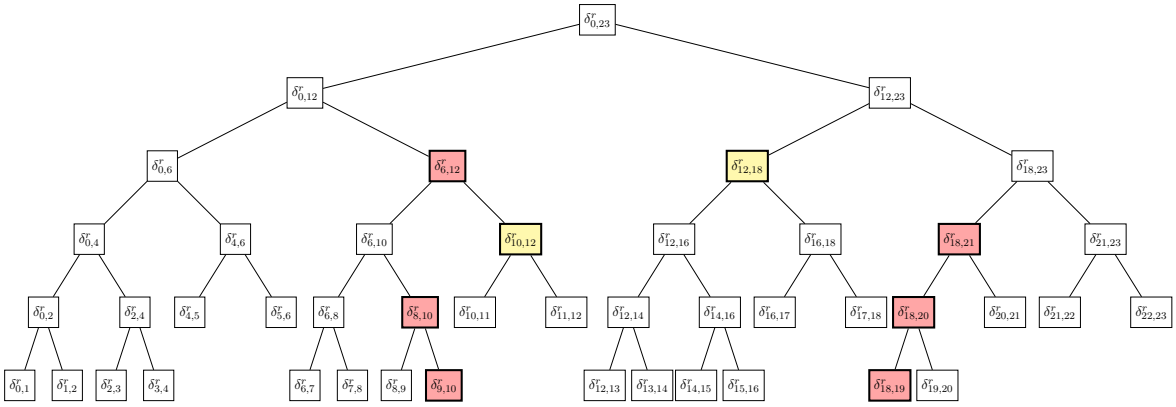

Figure 3: Counterexample A ($[\,10, 18\,)$): (L1) Downward adds out-of-interval nodes (red); correct nodes are yellow.

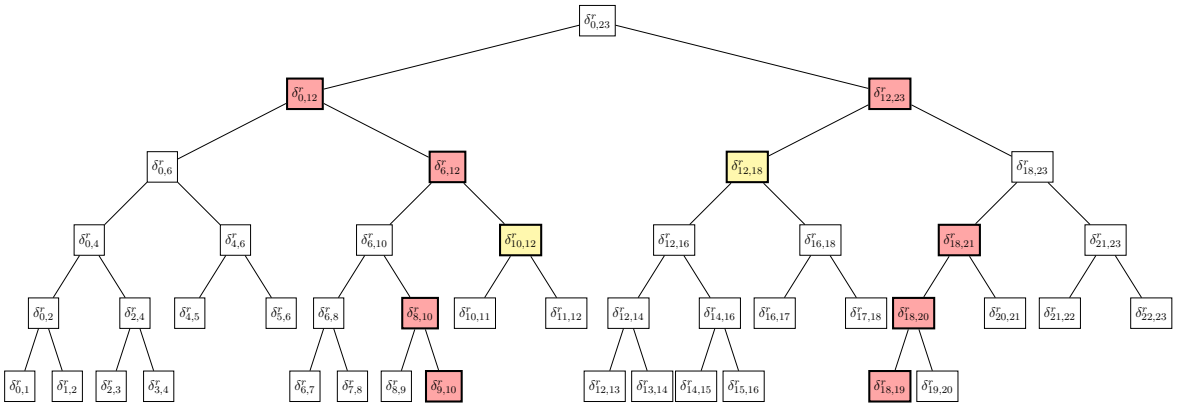

Figure 4: Counterexample A ($[\,10, 18\,)$): (L2) Upward adds out-of-interval nodes (red); correct nodes are yellow.

(L1) Downward. Along the left search path (toward $\delta^r_{9,10}$), these nodes are added: $\{\delta^r_{6,12}, \delta^r_{10,12}, \delta^r_{8,10}, \delta^r_{9,10}\}$. Along the right search path (toward $\delta^r_{18,19}$), these nodes are added: $\{\delta^r_{12,18}, \delta^r_{18,21}, \delta^r_{18,20}, \delta^r_{18,19}\}$. See Figure 3.

(L2) Upward. Along the left search path (toward split node $\delta^r_{0,23}$), these nodes are added: $\{\delta^r_{9,10}, \delta^r_{8,10}, \delta^r_{10,12}, \delta^r_{6,12}, \delta^r_{12,23}\}$. Along the right search path (toward split node $\delta^r_{0,23}$), these nodes are added: $\{\delta^r_{18,19}, \delta^r_{18,20}, \delta^r_{18,21}, \delta^r_{12,18}, \delta^r_{0,12}\}$. See Figure 4.

We present two additional counterexamples in Appendix C.

If the chain-retrieval algorithm returns an invalid chain, it increases the risk of discarding feasible or improving moves. Conversely, if a chain is valid but not minimal, the extra nodes impose unnecessary overhead due to performing more compositions than necessary. In Counterexample A, for the query $[10, 18)$, the chain $\{\delta^r_{16,18},\ \delta^r_{12,16},\ \delta^r_{10,12}\}$ is valid, yet not minimal. This overhead becomes even more pronounced for *generalized* routes in which each customer request has a set of candidate service locations, each with its own time window, leading to tree nodes that store a vector of CPL ready-time functions (see our companion work (Bornay, Gendreau, and Gendron 2025)).

These counterexamples confirm that a new, provably correct algorithm is required.

4 Provably Optimal Retrieval Algorithms

Having established the shortcomings of the existing retrieval rule, we now present two robust algorithms that correctly solve the minimal-chain retrieval problem on any query $[\,i, k\,)$ that is not an explicit node of the tree. The first is a top-down *Canonical Decomposition Search* (CDS), representing the standard and correct segment-tree approach. The second is a novel, bottom-up *Bidirectional Binary-Tree Search* (BBTS) tailored to this application. We prove that both algorithms are *valid* (satisfy alignment and containment), *minimal* (return the unique minimal chain), and *efficient* (time and space $\Theta(\log n)$ on balanced trees). For completeness we include a *Corrected Split–Path Retrieval* (CSPR), a repair of the rule in Visser and Spliet (2020); its correctness, minimality, and $\Theta(\log n)$ cost are proved in Appendix B (Thm. 16, Prop. 17).

Scope and equivalence. CDS (top-down), BBTS (bottom-up), and CSPR (repaired split-path) are *implementation-distinct* procedures that, for any fixed query $[\,i,k\,)$, return the same inclusion-maximal cover (Def. 9). Their outputs coincide since the minimal chain (Def. 6) is unique (Cor. 11), but their traversals, data needs, and natural output orders differ.

Convention. Intervals are half-open: $[\,i,k\,) = [i,k)$ with $i < k$. Leaves represent $\delta^r_{t,t+1}$, so the *inner* boundary leaves of $[\,i,k\,)$ are $\delta^r_{i,i+1}$ and $\delta^r_{k-1,k}$. (Outer neighbors $\delta^r_{i-1,i}$ and $\delta^r_{k,k+1}$ are also defined at the ends of $\sigma(r)$ when needed.)

We provide required definitions and a proposition that, together with prior lemmas, support our optimality proofs.

Definition 8 (Inclusion-maximal w.r.t. $[\,i,k\,)$) *A node $\delta^r_{a,b}$ is* inclusion-maximal in $[\,i,k\,)$ *if $[\,a,b\,) \subseteq [\,i,k\,)$ and there is no node $\delta^r_{c,d}$ with $[\,a,b\,) \subset [\,c,d\,) \subseteq [\,i,k\,)$.*

Definition 9 (Maximal cover) *A set $\mathcal{S}$ of nodes is a* maximal cover of $[\,i,k\,)$ *if (i) the intervals $\{[\,a,b\,) : \delta^r_{a,b} \in \mathcal{S}\}$ are pairwise* interior-disjoint *(overlap only at shared endpoints), (ii) their union equals $[\,i,k\,)$, and (iii) every $\delta^r_{a,b} \in \mathcal{S}$ is inclusion-maximal in $[\,i,k\,)$.*

Proposition 10 (Minimal chain $\iff$ maximal cover) *For any query $[\,i,k\,)$, a valid chain (Def. 6) is minimal iff its nodes form a maximal cover of $[\,i,k\,)$.*

Proof. ($\Rightarrow$) If a chain is minimal and some node were not inclusion-maximal in $[\,i,k\,)$, it could be replaced (together with sibling(s) sharing its parent) by a strictly larger node inside $[\,i,k\,)$, reducing the count - contradiction. Interior disjointness and full cover follow from Lemma 5. ($\Leftarrow$) If a cover is inclusion-maximal and interior-disjoint, no two nodes can be siblings with a parent inside $[\,i,k\,)$; otherwise (iii) is violated. Hence no coarsening is possible, so the chain is minimal. □

Corollary 11 (Uniqueness of the minimal chain) *For any query $[\,i,k\,)$, the minimal chain is unique.*

Proof. By Proposition 10, a minimal chain corresponds to a maximal cover of $[\,i,k\,)$. In a full segment tree, the left-to-right greedy choice of the largest node starting at each uncovered boundary is forced, hence the cover (and thus the chain) is unique. □

Algorithm 1: Canonical Decomposition Search (Iterative)

```
Function canonical_decomposition_search(query_start, query_end)
    Input: Integers with query_start < query_end; interval [query_start, query_end)
    Output: List of nodes forming the canonical decomposition
    exact_node ← iterative_search(query_start, query_end);
    if exact_node ≠ NULL then
        return {exact_node}
    result ← empty list;
    find_canonical_nodes_iterative(root, query_start, query_end, result);
    return result

Function find_canonical_nodes_iterative(root, query_start, query_end, result)
    if root = NULL then
        return
    S ← new Stack; S.push(root);
    while S not empty do
        node ← S.pop();
        if node.end ≤ query_start or node.start ≥ query_end then
            continue // Case 1:  No overlap for half-open intervals
        if node.start ≥ query_start and node.end ≤ query_end then
            add node to result;
            continue // Case 2:  Full containment
        if node.right ≠ NULL then
            S.push(node.right)
        if node.left ≠ NULL then
            S.push(node.left)
        // Case 3:  Partial overlap
```

4.1 Canonical Decomposition Search (CDS): Top-Down

The standard approach to finding a minimal set of nodes covering an interval in a segment tree is a top-down search (canonical decomposition / 1D range query (De Berg et al. 2008)). The algorithm traverses from the root, recursively exploring branches that overlap the query, pruning subtrees with no overlap, and collecting nodes fully contained in it. We present both recursive and iterative implementations; the *iterative* form avoids deep call stacks.

Theorem 12 (Correctness and Optimality of CDS) *For any query $[\,i, k\,)$ with no explicit node $\delta^r_{i,k}$ in the tree, CDS (Alg. 1 and Alg. 2) returns the unique minimal chain.*

Proof. Validity. By its pruning rules for half-open intervals, CDS only adds nodes fully contained in $[\,i, k\,)$; selected intervals are interior-disjoint and their union is $[\,i, k\,)$ (Lemma 5).

Inclusion-maximality. If a returned $\delta^r_{a,b}$ were not inclusion-maximal (Def. 8), a strict ancestor $\delta^r_{c,d}$ with $[\,a, b\,) \subset [\,c, d\,) \subseteq [\,i, k\,)$ would have been selected at the moment of full containment, contradicting the presence of its child. Thus the output is a maximal cover (Def. 9); by Prop. 10 the chain is minimal and, by Cor. 11, unique. □

Proposition 13 (Cost of CDS) *CDS runs in time $\Theta(\log n)$ and auxiliary space $\Theta(\log n)$, where n is the number of leaves.*

Proof. At most two root–to–leaf paths of length $h = \Theta(\log n)$ are touched; the number of selected nodes is $m \leq 2h$. Work is $\Theta(h + m) = \Theta(\log n)$; space is $\Theta(h)$ (stack frames or an explicit stack). □

Algorithm 2: Canonical Decomposition Search (Recursive)

```
Function canonical_decomposition_search_recursive(query_start, query_end)
    Input: Integers with query_start < query_end; interval [query_start, query_end)
    Output: List of nodes forming the canonical decomposition
    exact_node ← iterative_search(query_start, query_end);
    if exact_node ≠ NULL then
        return {exact_node}
    result ← empty list;
    find_canonical_nodes_recursive(root, query_start, query_end, result);
    return result

Function find_canonical_nodes_recursive(node, query_start, query_end, result)
    if node = NULL then
        return
    if node.end ≤ query_start or node.start ≥ query_end then
        return
    // No overlap
    if node.start ≥ query_start and node.end ≤ query_end then
        add node to result;
        return // Full containment
    find_canonical_nodes_recursive(node.left, query_start, query_end, result);
    find_canonical_nodes_recursive(node.right, query_start, query_end, result);
```

4.2 Bidirectional Binary-Tree Search (BBTS): Bottom-Up

As an alternative to top–down, we design a bottom–up bidirectional search that starts from the *inner* boundary leaves $\delta^r_{i,i+1}$ and $\delta^r_{k-1,k}$, ascends to their lowest common ancestor (the *split*), and builds the chain *inward* from both sides. BBTS (Alg. 3) trims nodes above the split, then grows two lists: a left list $\mathcal{L}$ that advances a *left frontier* from i rightward and a right list $\mathcal{R}$ that advances a *right frontier* from

Algorithm 3: Bidirectional Binary-Tree Search (BBTS) for $[\,i,k\,)$ with $i < k$

Input: Full binary tree $\mathscr{T}$ over leaves $\delta^r_{t,t+1}$; query interval $[\,i,k\,)$ (half-open).
Output: Queue $\mathcal{Q}$ of nodes forming the minimal chain for $[\,i,k\,)$.

Init and trivial cases:
if $k = i+1$ **then**
  return queue$[\mathscr{T}.\text{leaves}[i]]$
if $\exists$ *explicit node* $\delta^r_{i,k}$ *in* $\mathscr{T}$ **then**
  return queue$[\delta^r_{i,k}]$
Step 1: Inner boundary leaves
left_adj $\leftarrow \mathscr{T}.\text{leaves}[i]$, *right_adj* $\leftarrow \mathscr{T}.\text{leaves}[k-1]$;
Step 2: Find split (LCA) and trim paths
$P_L \leftarrow \text{path\_to\_root}(\textit{left\_adj})$; $P_R \leftarrow \text{path\_to\_root}(\textit{right\_adj})$;
split $\leftarrow \text{first\_common\_node}(P_L, P_R)$;
if $\text{interval}(\textit{split}) = [\,i,k\,)$ **then**
  return queue[*split*]
Trim P_L and P_R to exclude nodes *above split*;
Step 3: Grow maximal cover from both sides
$\mathcal{L} \leftarrow \text{search\_left}(P_L, i, k)$; $\mathcal{R} \leftarrow \text{search\_right}(P_R, i, k)$;
Step 4: Merge left→right
return $\text{merge}(\mathcal{L}, \mathcal{R})$;

Helper: left side (advance a left frontier)
Function `search_left(`P_L, i, k`):`
  $\mathcal{L} \leftarrow [\,]$; $\ell \leftarrow i$; `// current left frontier`
  foreach x *on the* ascend *from left_adj to split (exclude split)* **do**
    if x *is the* left *child of its parent* p **then**
      $u \leftarrow \text{Right}(p)$; `// candidate starts at` ℓ
      if $\text{interval}(u) \subseteq [\,i,k\,)$ *and* $\text{start}(u) = \ell$ **then**
        while u *has right child* u_r *with* $\text{interval}(u_r) \subseteq [\,i,k\,)$ *and* $\text{start}(u_r) = \ell$ **do**
          $u \leftarrow u_r$; `// greedily enlarge starting at` ℓ
        append u to $\mathcal{L}$; $\ell \leftarrow \text{end}(u)$;
  return $\mathcal{L}$;

Helper: right side (advance a right frontier)
Function `search_right(`P_R, i, k`):`
  $\mathcal{R} \leftarrow [\,]$; $r \leftarrow k$; `// current right frontier`
  foreach y *on the* ascend *from right_adj to split (exclude split)* **do**
    if y *is the* right *child of its parent* q **then**
      $v \leftarrow \text{Left}(q)$; **if** $\text{interval}(v) \subseteq [\,i,k\,)$ *and* $\text{end}(v) = r$ **then**
        while v *has left child* v_ℓ *with* $\text{interval}(v_\ell) \subseteq [\,i,k\,)$ *and* $\text{end}(v_\ell) = r$ **do**
          $v \leftarrow v_\ell$; `// greedily enlarge ending at` r
        prepend v to $\mathcal{R}$; $r \leftarrow \text{start}(v)$;
  return $\mathcal{R}$;

k leftward. A side accepts a node iff it is contained in $[\,i, k\,)$ and exactly adjoins the current frontier. Finally, BBTS merges $\mathcal{L}$ and $\mathcal{R}$ (preserving order) to obtain the chain.

Theorem 14 (Correctness and Optimality of BBTS) *For any query $[\,i, k\,)$ with no explicit node $\delta^r_{i,k}$, BBTS (Alg. 3) returns the unique minimal chain.*

Proof. Validity (cover + interior-disjointness). BBTS maintains two invariants. (L) The left list $\mathcal{L}$ covers a contiguous prefix $[\,i, \ell\,)$ with interior-disjoint nodes, where ℓ is its current frontier; each accepted node begins at ℓ and lies within $[\,i, k\,)$. (R) Symmetrically, the right list $\mathcal{R}$ covers a contiguous suffix $[\,r, k\,)$ with interior-disjoint nodes, where r is its current frontier; each accepted node ends at r and lies within $[\,i, k\,)$. Both invariants hold initially ($\ell = i$, $r = k$) and are preserved because BBTS only appends a node when it is fully contained in $[\,i, k\,)$ and exactly adjoins the corresponding frontier. As the paths meet at the split, $\mathcal{L}$ and $\mathcal{R}$ remain disjoint and, after merging, cover $[\,i, k\,)$ without gaps (Lemma 5).

Inclusion-maximality ⇒ minimality. At each acceptance, BBTS greedily enlarges to the largest node that starts (left side) or ends (right side) at the current frontier while staying inside $[\,i, k\,)$; any larger ancestor would have been encountered earlier. Thus every accepted node is inclusion-maximal in $[\,i, k\,)$, so the output is a maximal cover (Def. 9). By Prop. 10 the chain is minimal; by Cor. 11 it is unique. □

Proposition 15 (Cost of BBTS) *BBTS runs in time $\Theta(\log n)$ and auxiliary space $\Theta(\log n)$, where n is the number of leaves.*

Proof. Tracing the two inner boundary leaves to the root, finding the LCA, and descending along trimmed paths touch at most $O(h)$ nodes per side, with $h = \Theta(\log n)$ the height. The number of accepted nodes satisfies $m \le 2h$, so the total work is $\Theta(h + m) = \Theta(\log n)$. The algorithm stores $O(h)$ nodes for the paths and the partial lists $\mathcal{L}, \mathcal{R}$, hence $\Theta(\log n)$ space. □

5 Computational Experiments

We build balanced trees $\mathscr{T}^r$ for route sequences $\sigma(r) = \{0, 1, \ldots, n\}$ for $n \in \{14, 24, 32, 40, 50, 64, 80, 100\}$. Leaves represent $\delta^r_{t,t+1}$ and queries are $\delta^r_{i,k}$ over half-open intervals $[i, k)$ with $0 \le i < k \le n$. For each tree we evaluate every query, using 10 warmups and 100 timed repetitions per query. We record runtime in nanoseconds, chain length ℓ, visited nodes, and an approximate byte count (#visited × node size).

Algorithms. We compare six correct methods: {I-CSPR, R-CSPR, I-BBTS, R-BBTS, I-CDS, R-CDS}. CSPR is the corrected split-path retrieval; CDS is the canonical decomposition (segment-tree) search; BBTS is our bottom-up bidirectional search. All return the same minimal chain for any query $[i, k)$. Unless stated otherwise, **I-CSPR** is the baseline.

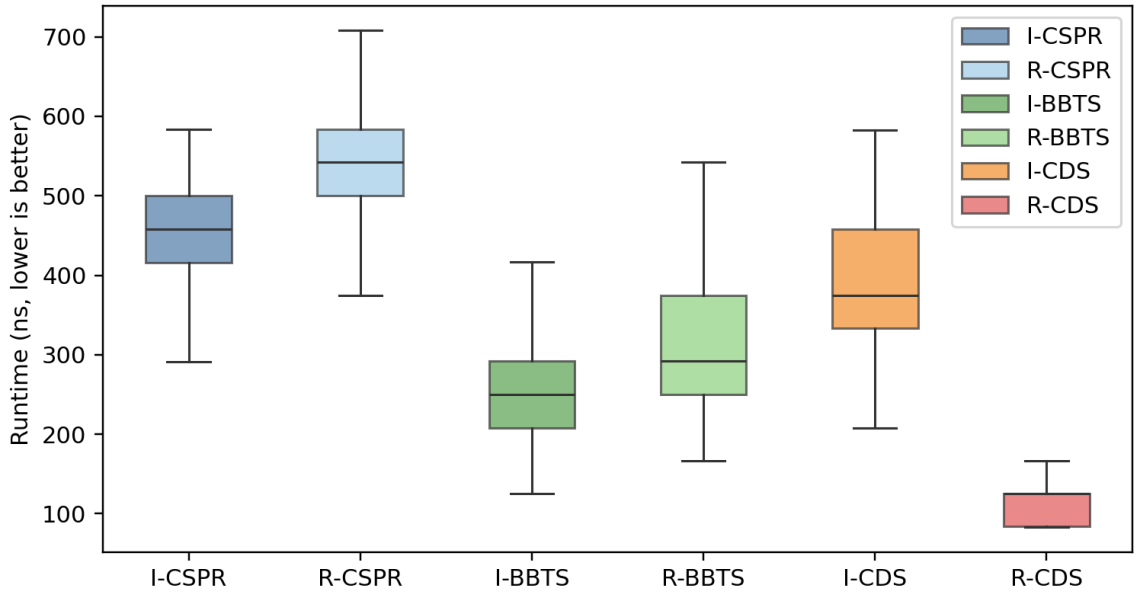

Figure 5: Per-query runtime on the cover set, aggregated over all trees. Lower is better.

5.1 Results

We report the runtime of algorithms, number of visited nodes, and the approximate memory traffic.

Runtime (Figure 5). Across all trees, the ordering is stable:

$$\text{R-CDS (fastest)} \;<\; \text{I-BBTS} \;<\; \text{R-BBTS} \approx \text{I-CDS} \;<\; \text{I-CSPR} \;<\; \text{R-CSPR (slowest)}.$$

R-CDS shows the lowest medians and tight dispersion; I-BBTS is the next best and clearly outperforms the split-path variants.

Visited nodes (Figure 6). Work counters mirror the timing: CSPR variants visit the most nodes; BBTS the fewest; CDS is in between. I-BBTS typically achieves the smallest visit counts, while R-CDS attains the best time/visit trade-off.

Bytes (Figure 7). The approximate memory traffic follows the same pattern as visited nodes, supporting that speedups come from less traversal rather than measurement noise.

All methods are provably correct; for speed we recommend **R-CDS** as the default, with **I-BBTS** a strong non-recursive alternative.

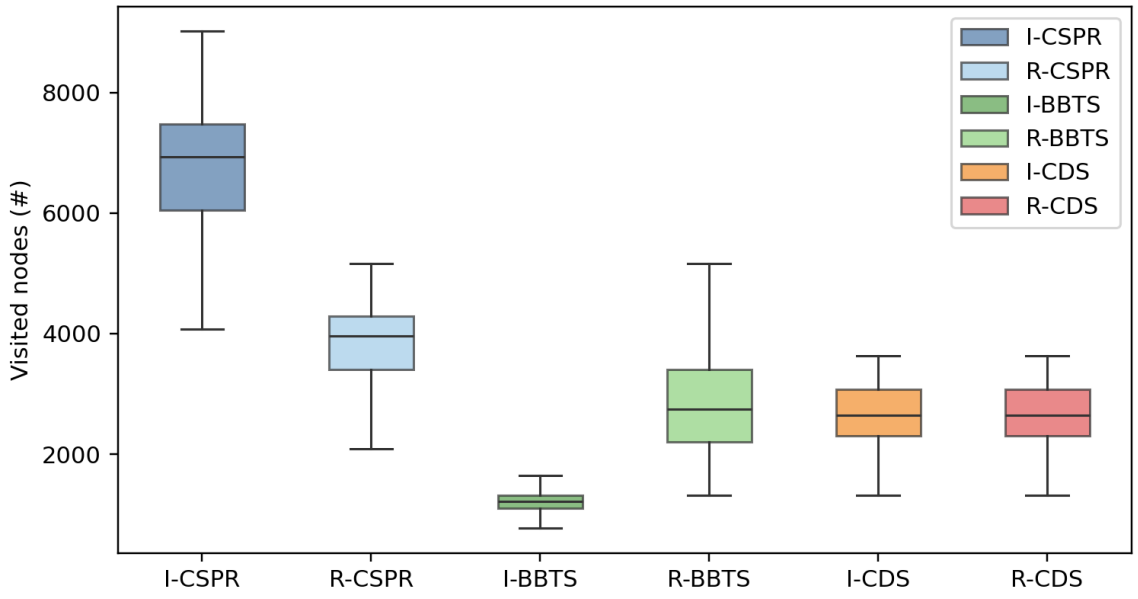

Figure 6: Visited nodes (#) on the cover set, aggregated over all trees.

5.2 Evaluation of Minimal-Chain Retrieval within a Sequential Route Construction Heuristic

We assess the impact of minimal-chain retrieval on end-to-end heuristic runtime by embedding each retrieval algorithm into a sequential route construction heuristic (SRCH) for a Generalized Dial-a-Ride Problem instance with $3{,}000$ requests and time-dependent road travel times. The fleet is homogeneous and each constructed route is scheduled to optimality under a route-duration objective.

Schedulers and precomputation. We use two exact route schedulers: *Recursive Post-order Binary Tree Propagation* (R-PO-BTP) and its *iterative* counterpart (I-PO-BTP). These DPs are invoked *only when an insertion is accepted*—not during the offline evaluation of candidate moves. When a DP is called, it computes and stores lower envelopes of optimal (extended) departure-time functions $\delta^{\tau}_{a,b}$ in a matrix for all explicit nodes in the tree, i.e., for half-open intervals $[a, b)$ represented by tree nodes.

Role of minimal-chain retrieval. During the SRCH inner loop, evaluating a candidate insertion requires repeated queries $\delta^{\tau}_{i,k}$ over arbitrary $[i, k)$ that are *not* necessarily explicit nodes. To avoid re-running the DP for each candidate position, we compose the already stored envelopes via a minimal chain of explicit nodes whose union is exactly $[i, k)$. The only component that varies across treatments is the retrieval routine that returns this chain; the downstream cost computation and acceptance logic are identical.

For further details on the SRCH and scheduling framework, see Bornay, Gendreau, and Gendron (2025).

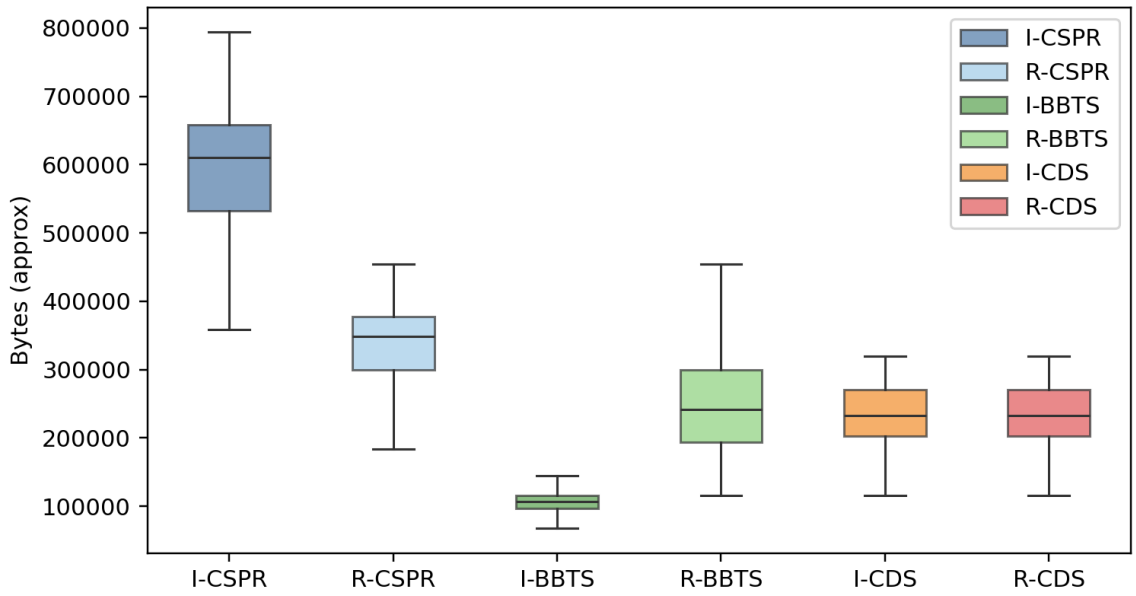

Figure 7: Approximate bytes touched on the cover set, aggregated over all trees.

We test the six correct retrieval algorithms from Section 4:

$$\{\textbf{I-CSPR},\ \textbf{R-CSPR},\ \textbf{I-BBTS},\ \textbf{R-BBTS},\ \textbf{I-CDS},\ \textbf{R-CDS}\}.$$

Each is paired with each scheduler (R-PO-BTP or I-PO-BTP), yielding 2×6 SRCH variants. **I-CSPR** serves as the baseline.

Since all six retrieval methods return the same unique minimal chain for any query $[i, k)$, solution quality (objective value, feasibility) is invariant across treatments. We therefore focus on performance metrics: (i) total SRCH wall-clock time, (ii) inner-loop throughput (evaluations per second), (iii) distributional summaries of per-query time and visited-node and byte counters collected during SRCH, and (iv) fraction of SRCH time spent inside retrieval versus other components (DP calls, insertion bookkeeping). Hardware and compiler details, instance generation, and repetition counts are identical across treatments.

Expectation. Given the microbenchmarks in Section 5, we expect **R-CDS** to yield the fastest SRCH runs, with **I-BBTS** a strong non-recursive alternative; CSPR variants are anticipated to be slower due to larger visit/byte footprints. The experiment isolates these effects in an application-relevant setting.

Results and discussion. Embedding minimal-chain retrieval into SRCH reduces end-to-end runtime relative to the I–CSPR baseline. On the 3,000-request instance, median speedups (p95 in caps) are: R-BBTS $+3.04\%$ (3.20), I-CDS $+2.88\%$ (3.05), R-CDS $+2.41\%$ (2.65), I-BBTS $+1.76\%$ (1.95), R-CSPR $+1.47\%$ (1.76). The small median - p95 gaps (0.16 - 0.29 percentage points) indicate stable

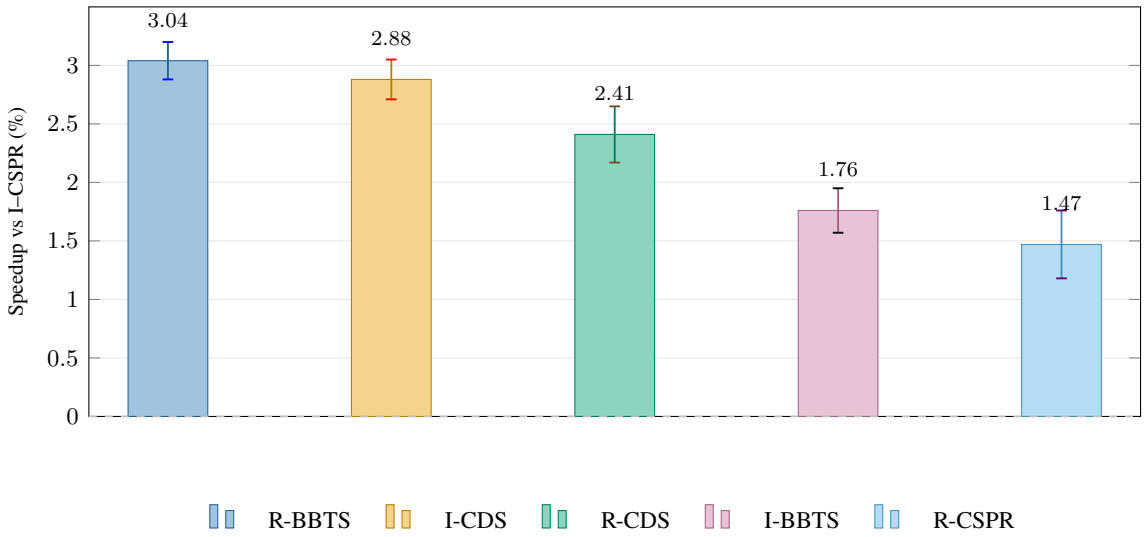

Figure 8: Median SRCH wall-clock speedup of minimal-chain retrieval algorithms relative to I–CSPR. Bars are medians; caps indicate the 95th percentile across repeats (higher is better).

rankings across repeats. The ordering matches the work counters (visited nodes, bytes), confirming that BBTS/CDS accelerate SRCH by reducing traversal rather than exploiting timing noise. Because retrieval is invoked at every candidate evaluation, these per-query savings compound over tens of thousands of queries, yielding material wall-clock reductions with identical schedule quality. Practically, **R-CDS** is a strong default, with **I-BBTS** a competitive non-recursive alternative.

These gains compound over the inner loop, so even single-digit percentages translate into meaningful reductions in total heuristic time without altering solution quality. Combined with $\Theta(\log n)$ complexity and a small code footprint, BBTS/CDS offer a practical upgrade path for existing TDVRP implementations.

Minimal-chain retrieval via BBTS/CDS is a strict improvement over (corrected) CSPR inside SRCH, delivering consistent end-to-end speedups while preserving optimal per-route schedules; we therefore recommend **R-CDS** by default (and **I-BBTS** when a non-recursive routine is preferred) for move evaluation and for offline exact route cost computation in time-dependent routing.

6 Conclusion

In this paper, we addressed a critical issue in the use of binary composition trees for accelerating move evaluations in time-dependent vehicle routing. We identified an error in the state-of-the-art retrieval algorithm proposed by Visser and Spliet (2020), demonstrating via counterexamples that their method can return composition chains that are either invalid or non-minimal. A reliable retrieval mechanism is fundamental to the integrity of any neighborhood-search heuristic that relies on this data structure.

To resolve this, we presented two robust, provably optimal algorithms: a top-down Canonical Decomposition Search (CDS), adapted from the standard method in computational geometry, and a

novel, bottom-up Bidirectional Binary-Tree Search (BBTS). We also provided a corrected split–path retrieval (CSPR) that repairs the rule of Visser and Spliet (2020) (Appendix B). For all three algorithms, we proved correctness and minimality, ensuring they consistently return the unique minimal valid chain for any query, and we established optimal $\Theta(\log n)$ time and auxiliary space bounds.

The primary contribution of this work is to solidify the foundation of the Ready Time Function Tree, making it a reliable and powerful tool for the operations-research community. Our algorithms can be integrated directly into offline move-evaluation frameworks, ensuring accurate cost calculations and enabling the efficient use of advanced, non-lexicographic neighborhood-search strategies. This paper thus serves both as a necessary correction to the literature and as a practical guide for implementing this key procedure in time-dependent routing.

The immediate importance of a correct retrieval mechanism is demonstrated in our companion work, where it is an essential component for solving a generalized route-scheduling problem with multiple candidate locations and network-level time dependency. We prove this generalized problem to be $\mathcal{NP}$-hard in the strong sense, reinforcing the need for efficient methods to exactly evaluate moves during solution improvement. Our fast, provably optimal retrieval algorithms are therefore critical: they provide the reliable subroutine required to make exact move evaluation practical for this complex problem class, without resorting to re-invoking the full dynamic-programming schedulers.

References

Adamo T, Gendreau M, Ghiani G, Guerriero E, 2024 *A review of recent advances in time-dependent vehicle routing*. *European Journal of Operational Research* .

Bornay B, Gendreau M, Gendron B, 2025 *Generalized route scheduling problem with time-dependent travel times on road networks*. Working Paper CIRRELT-2025-30, Interuniversity Research Centre on Enterprise Networks, Logistics and Transportation (CIRRELT), URL `https://cirrelt.ca/documentstravail/cirrelt-2025-30.pdf`.

De Berg M, Cheong O, Van Kreveld M, Overmars M, 2008 *Computational geometry: algorithms and applications* (Springer).

Ichoua S, Gendreau M, Potvin JY, 2003 *Vehicle dispatching with time-dependent travel times*. *European Journal of Operational Research* URL `http://dx.doi.org/10.1016/S0377-2217(02)00147-9`.

Visser TR, Spliet R, 2020 *Efficient move evaluations for time-dependent vehicle routing problems*. *Transportation science* 54(4):1091–1112.

Appendices

A Theorem 4 of Visser and Spliet (2020)

Theorem 4. *Given the ready time function tree $\mathcal{T}^r$ of a route r, any partial route ready time function $\delta^r_{i,j}$ can be obtained in at most $\mathcal{O}(\bar{m}p)$ operations, with $\bar{m}$ the number of vertices between i and j,*

inclusive, on route r.

Proof. Suppose $\delta^r_{i,j}$ needs to be obtained using the tree $\mathcal{T}^r$, with $\bar{m}$ vertices between i and j in route r. By general properties of balanced search trees (see de Berg et al. 2008, pp. 96–99), the ready time functions stored in the nodes of $\mathcal{T}^r$ that are most efficient for composing $\delta^r_{i,j}$ (the thick nodes in Figure 1) can be found as follows. Leaf node $\delta^r_{i-1,i}$ in $\mathcal{T}^r$ corresponds to the node directly left of the required interval and leaf node $\delta^r_{j,j+1}$ directly right next to the interval. Both these leaf nodes have a unique search path to the root node $\delta^r_{o,d}$. At some node, which we denote by δ^r_{split}, both search paths will be merged. In Figure 1, the leaf nodes are δ^r_{o1} and $\delta^r_{15,d}$ and their search path merges at $\delta^r_{\text{split}} = \delta^r_{o,d}$. The most efficient nodes in the tree for the composition can now be found to be all *right* child nodes of the nodes along the (left) search path of $\delta^r_{i-1,i}$ and all the *left* child nodes of the nodes along the (right) search path of $\delta^r_{j,j+1}$. Here, we denote the two children of a nonleaf node in the tree as being left or right, with the left child having the ready time function with lower indices. Let us denote the composition of the right children along the left search path by δ^L and likewise the composition of the left children among the right search path by δ^R. Now the required ready time function can be found by calculating the composition of these two parts: $\delta^r_{i,j} = (\delta^R \circ \delta^L)$. It can be proved (see de Berg et al. 2008, pp. 96–99) that for each level in the tree, at most two nodes with the same level will be part of the required composition. In case two nodes of the same level are present in the composition, one node will be contained in δ^L and the other must be in δ^R. Furthermore, nodes in the composition δ^L increase in level with larger indices, and the nodes in the composition δ^R decrease in level with larger indices. Therefore, given the interval $[i, j]$ of the required ready time function and its corresponding split node δ^r_{split}, the composition consists in the worst case of one node of each level below δ^r_{split} in δ^L and also one node of each level below δ^r_{split} in δ^R. There are $\bar{m}$ vertices between i and j, inclusive. Let $\bar{l}$ be highest level in the tree of which nodes can appear in the composition, that is, one level below the split node. It can be seen that $\bar{l} \leq \lceil \log(\bar{m} + 2) \rceil - 1$. We have seen that the order of evaluating compositions is most efficient when iteratively selecting the composition involving the smallest functions. Therefore, the compositions in δ^L are evaluated from the lowest-level nodes up to level $\bar{l}$, in Figure 1 from left to right, and likewise the compositions in δ^R are evaluated, in Figure 1 from right to left, and finally $\delta^r_{i,j} = (\delta^R \circ \delta^L)$ is evaluated. The total number of operations required to calculate $\delta^r_{i,j}$ in the worst case using this procedure is given by

$$
\begin{aligned}
&\mathcal{O}\left(2\sum_{l=1}^{\bar{l}-1}\sum_{k=0}^{l} 2^k p \;+\; 2\sum_{k=0}^{\bar{l}-1} 2^k p\right) \\
&= \mathcal{O}\left(2\sum_{l=2}^{\bar{l}} [(2^l - 1)]p \;+\; 2(2^{\bar{l}} - 1)p\right) \\
&= \mathcal{O}\Big(2(2^{\bar{l}+1} - 4 - \bar{l} - 1)p \;+\; 2(2^{\bar{l}} - 1)p\Big) \\
&= \mathcal{O}\Big(3 \cdot 2^{\bar{l}+1} p \;-\; (\bar{l} + 12)p\Big)
\end{aligned}
$$

$$= \mathcal{O}(3\bar{m}p)$$
$$= \mathcal{O}(\bar{m}p). \tag{2}$$

In Equation (2), the double summation in the first term represents the worst-case number of operations needed to construct δ^L and is counted twice for also constructing δ^R. The last term in Equation (2) corresponds to the worst-case number of operations needed to evaluate the composition $\delta^r_{ij} = (\delta^R \circ \delta^L)$. Since $\delta^r_{i,j}$ contains at most $\mathcal{O}(\bar{m}p)$ breakpoints, $\delta^r_{i,j}$ can be obtained from $\delta^r_{i,j}$ in at most $\mathcal{O}(\bar{m}p)$ operations. Therefore, using the tree $\mathcal{T}^r$ in memory, $\delta^r_{i,j}$ can be obtained in at most $\mathcal{O}(\bar{m}p)$ operations. □

B Corrected Split–Path Retrieval (CSPR)

For completeness we include a repaired split–path retrieval that is *output–equivalent* to CDS/BBTS but implementation–distinct. The procedure ascends from the *outer* neighbors of the query—leaves $[i-1, i)$ and $[k, k+1)$ when they exist—up to their lowest common ancestor (the *split*). While ascending, it collects *siblings* under the following guardrails:

- *Stop at split.* Both ascents stop when the parent is the split; we do not add the split's children unless a boundary exception applies.
- *Sibling–while–ascending.* On the left ascent, whenever the current node is a *left* child, collect its parent's *right* child; on the right ascent, whenever the current node is a *right* child, collect its parent's *left* child.
- *Containment filter.* Each collected sibling is added only if its interval lies within $[i, k)$; this prevents overshoot beyond the query (cf. Lemma 5).
- *Boundary exception.* If one side contributes nothing (e.g., the query touches a tree boundary so the corresponding outer neighbor is absent), we add the split's child on that side *iff* its interval is contained in $[i, k)$; this covers the otherwise missing flank.

Finally, the right–side list is reversed and concatenated to the left–side list, producing a left–to–right chain that respects adjacency (Lemma 4).

Theorem 16 (Correctness and minimality of CSPR) *For any query $[i, k)$ that is not an explicit node, Algorithm 4 (CSPR) returns a chain that is* valid *(satisfies Lemmas 4–5) and* minimal*. The returned chain equals the inclusion–maximal cover of $[i, k)$ (Def. 9) and thus coincides with the output of CDS and BBTS by Prop. 10 and Cor. 11.*

Proof. **Validity.** By construction, only intervals contained in $[i, k)$ are admitted (containment filter), and reversing the right collection yields a left–to–right sequence whose endpoints meet exactly at

Algorithm 4: Corrected Split-Path Retrieval (CSPR) for query $[i, k)$

```
Function CSPR(i, k)
    Input: Query interval [i, k) with i < k on tree 𝒯 (half-open).
    Output: Minimal chain covering [i, k) as a left-to-right list of nodes.
    /* (0) Fast path: explicit node present                                   */
    n* ← iterative_search(i, k);
    if n* ≠ NULL then
        return {n*};
    /* (1) Identify outer neighbors (may be absent at boundaries)             */
    u_L ← leaf for [i−1, i) if i > first else NULL;
    u_R ← leaf for [k, k+1) if k < last else NULL;
    /* (2) Find split (LCA) with boundary fallback                            */
    if u_L ≠ NULL and u_R ≠ NULL then
        split ← LCA(u_L, u_R);
    else
        /* If one neighbor is missing, use the root as a conservative split  */
        split ← root of 𝒯;
    L ← empty list; R ← empty list;
    /* (3) Ascend from the left outer leaf to (but not including) split       */
    if u_L ≠ NULL then
        cur ← u_L;
        while Parent(cur) ≠ split do
            if cur is Left (Parent(cur)) then
                x ← Right (Parent(cur));
                if interval (x) ⊆ [i, k) then
                    append x to L;
            cur ← Parent(cur);
    /* (4) Ascend from the right outer leaf to (but not including) split      */
    if u_R ≠ NULL then
        cur ← u_R;
        while Parent(cur) ≠ split do
            if cur is Right (Parent(cur)) then
                x ← Left (Parent(cur));
                if interval (x) ⊆ [i, k) then
                    append x to R;
            cur ← Parent(cur);
    /* (5) Boundary exceptions at the split (only when one side collected nothing) */
    if L is empty then
        x ← Right (split);
        if x ≠ NULL and interval (x) ⊆ [i, k) then
            append x to L;
    if R is empty then
        x ← Left (split);
        if x ≠ NULL and interval (x) ⊆ [i, k) then
            append x to R;
    /* (6) Merge then orient for composition (right-to-left)                  */
    reverse (R);
    return reverse ( L followed by R );
```

shared boundaries, enforcing adjacency (Lemma 4). The boundary exception guarantees full cover when one flank would otherwise be missing (Lemma 5).

Inclusion–maximality. Along each ascent, we add a sibling only when the current edge crosses from a child oriented *toward* $[i, k)$ (left on the left path, right on the right path). With the stop–at–split

rule, this selects, at each level below the split, the largest subtree completely contained in $[i, k)$ and adjacent to the yet–uncovered boundary. Any strictly larger node containing it would either violate containment or cross the split—both disallowed. Hence every chosen node is inclusion–maximal in $[i, k)$.

Minimality and uniqueness. Since the selected nodes are pairwise interior–disjoint and cover $[i, k)$, and each is inclusion–maximal, they form a maximal cover (Def. 9). By Prop. 10 this cover coincides with the minimal chain; by Cor. 11 it is unique and therefore matches CDS/BBTS. □

Proposition 17 (Cost of CSPR) *On a balanced tree with n leaves and height $h = \Theta(\log n)$, Algorithm 4 runs in $\Theta(h)$ time and uses $\Theta(h)$ auxiliary space.*

Proof. Let P_L and P_R be the two ascent paths from the outer neighbors to the split; each has length at most h. At each level, the algorithm examines at most one sibling per side and appends it only if its interval lies within $[i, k)$. Thus at most one node per level per side is collected and $|L| + |R| \leq 2h$. Reversing R and concatenating with L costs $O(|L| + |R|) = O(h)$; the two ascents and containment checks are also $O(h)$. The temporary working lists L and R therefore have size $O(h)$, yielding $\Theta(h)$ auxiliary space. Summing the costs gives overall $\Theta(h) = \Theta(\log n)$ time. □

C More Counterexamples

In what follows, we provide two additional counterexamples.

Counterexample B (boundary incompleteness).

Query $[\,i, j\,) = [\,0, 15\,)$. The left search path is empty. Under (L1), omitting the root's left child leaves a gap on $[\,0, 8\,)$ (red node in Figure 9). The correct minimal chain is

$$\{\delta^r_{0,8},\ \delta^r_{8,12},\ \delta^r_{12,14},\ \delta^r_{14,15}\} \xrightarrow{\text{leading to}} \delta^r_{0,15} = \delta^r_{14,15} \circ \delta^r_{12,14} \circ \delta^r_{8,12} \circ \delta^r_{0,8}.$$

Under (L2), ascending the right search path yields, in order, $\delta^r_{14,15}$, $\delta^r_{12,14}$, $\delta^r_{8,12}$, and $\delta^r_{0,8}$, which is valid for this boundary case (yellow nodes in Figure 9).

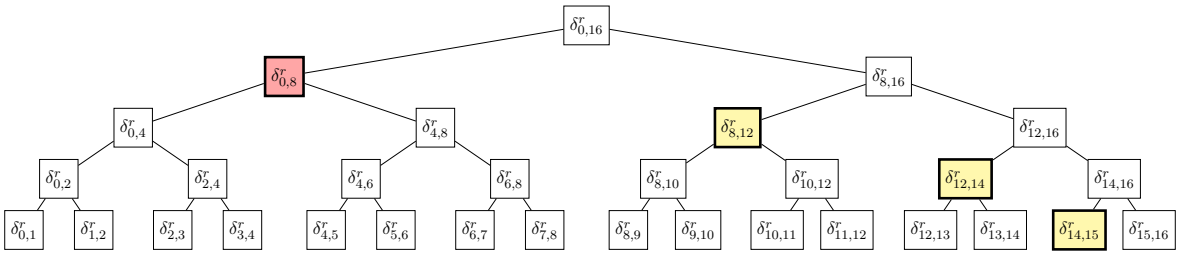

Figure 9: Counterexample B ($[\,0, 15\,)$): (L1) Downward with empty left search path; hence the node $\delta^r_{0,8}$ is missing. Query $\delta^r_{0,15}$ over $[\,0, 15\,)$.

Counterexample C (interior query; both L1 and L2 invalid).

Query $[\,i, j\,) = [\,2, 13\,)$.

(L1) Downward. The left descent collects $\{\delta^r_{4,8},\ \delta^r_{2,4},\ \delta^r_{1,2}\}$ (outside $[\,2, 13\,)$); the right descent collects $\{\delta^r_{8,12},\ \delta^r_{12,14}\}$ (overshoots j=13) and *misses* $\delta^r_{12,13}$. Thus L1 violates Lemma 5 (containment), and no ordering of the returned nodes can equal $\delta^r_{2,13}$ without either a gap or an overshoot.

(L2) Upward. Ascending from both sides and adding "all right children on the left path" and "all left children on the right path" yields out-of-interval nodes. From the left path one obtains $\{\delta^r_{2,4},\ \delta^r_{4,8},\ \delta^r_{8,16}\}$ (overshoot), and from the right path one obtains $\{\delta^r_{12,13},\ \delta^r_{8,12},\ \delta^r_{0,8}\}$ (extends left of i=2). Under the literal wording, the selections also include $\delta^r_{1,2}$ (right child of $\delta^r_{0,2}$ on the left path) and $\delta^r_{12,14}$ (left child of $\delta^r_{12,16}$ on the right path), both outside $[\,2, 13\,)$. Hence L2 likewise violates Lemma 5 (containment).

Minimal chain.

$$\{\delta^r_{2,4},\ \delta^r_{4,8},\ \delta^r_{8,12},\ \delta^r_{12,13}\} \xrightarrow{\text{leading to}} \delta^r_{2,13} = \delta^r_{12,13} \circ \delta^r_{8,12} \circ \delta^r_{4,8} \circ \delta^r_{2,4}.$$

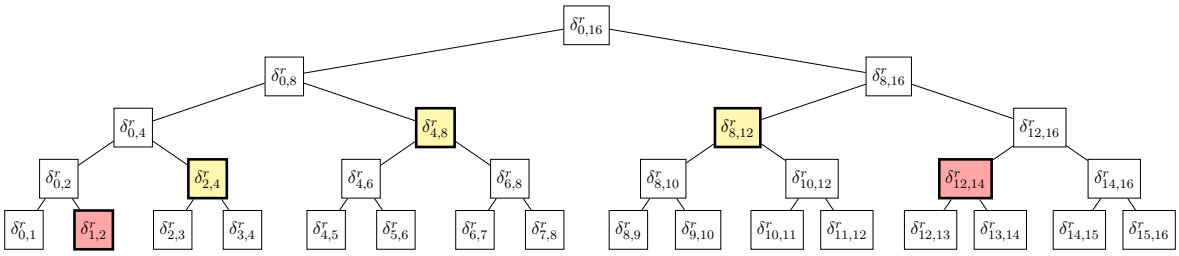

Figure 10: Counterexample C ($[\,2, 13\,)$): (L1) Downward collects out-of-interval nodes $\delta^r_{1,2}$ and $\delta^r_{12,14}$ (red) and *misses* $\delta^r_{12,13}$; yellow nodes are the correct ones for a valid chain.

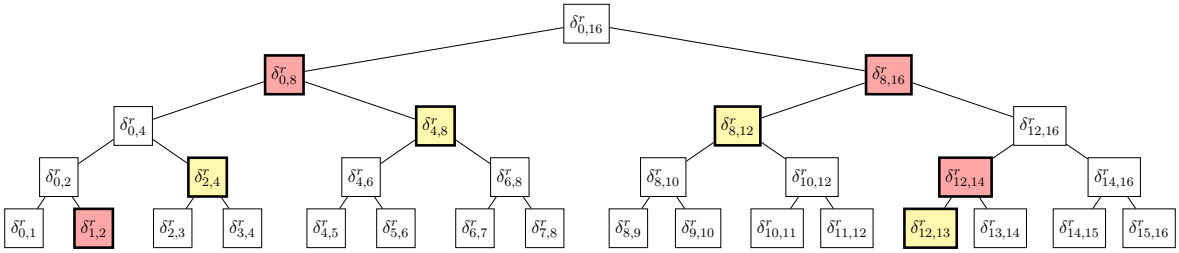

Figure 11: Counterexample C ($[\,2, 13\,)$): (L2) Upward adds out-of-interval nodes $\delta^r_{8,16}$ and $\delta^r_{0,8}$ (red); correct nodes are yellow.