MAMUT-routing Standard Specification

Numerical semantics for static and time-dependent routing benchmarks

Adrien Pichon[1*] Florian Rascoussier[2,3*]
Romain Billot[2†] Lina Fahed[2†] Alexandru-Liviu Olteanu[1†]
Marc Sevaux[1†] Christine Solnon[3†]

[1]Université Bretagne Sud, Lab-STICC, France
[2]IMT Atlantique, Lab-STICC, France
[3]INSA Lyon, Inria, CITI, France

Version 0.1, November 26, 2026

Abstract

We specify how a MAMUT-routing benchmark edition assigns numerical meaning to an instance, a route and its reported cost. The instance manifest binds the source artifacts by hash, so that a reader can either load the published arc data or reproduce them with pinned code and compare their complete binary64 digest. Once these operands are fixed, both readers can evaluate the submitted routes exactly. Static benchmarks use a cost matrix, whereas time-dependent benchmarks use arrival-time functions and optimize the departure of each route. In both settings, we compare incumbents using the exact objective and derive the displayed values from it. We give the shared contracts, the evaluation rules specific to each setting and worked examples of their consequences.

Author	ORCID identifier
Adrien Pichon	0009-0005-8630-3962
Florian Rascoussier	0009-0005-3253-9814
Romain Billot	0000-0002-2614-3822
Lina Fahed	0000-0002-7207-0580
Alexandru-Liviu Olteanu	0000-0002-0715-2168
Marc Sevaux	0000-0003-3855-9072
Christine Solnon	0000-0002-0919-496X

*First authors, listed alphabetically by last name.
†Second authors, listed alphabetically by last name.

Contents

1 Purpose, scope and conformance 4

2 From JSON numbers to exact operands 6

3 Four contracts for a benchmark result 8

4 Identify and materialize the instance 8

5 Feasibility and cost of a submitted solution 11

5.1 Which customers does each vehicle serve? 11

5.2 From depot departure to depot return 12

5.3 Choosing the depot departure in the TD problem 13

5.4 From route values to the solution objective 14

6 From a checked route to a stored result 15

6.1 The exact value and its display 15

6.2 What the TD witness adds 16

6.3 Record identity and compare like with like 17

6.4 Rank a candidate against the incumbent 18

References 18

A Mathematical notation and arithmetic domains 21

A.1 Number sets and their relationships 21

A.2 Parsing, value preservation and rounding 22

A.3 Symbols and notation 23

B Canonical byte streams for materialized arc data 26

C Implementing the two exact binary64 profiles **28**

C.1 Static routes on a common integer scale 28

C.2 TD routes with arbitrary rationals 30

D Alternative input readings and arithmetic modes **32**

D.1 Integral and decimal readings 32

D.2 Rounded and historical evaluation 33

E Attainment of the minimum route duration **34**

F A departure at a jump of the return arc **35**

G Solver self-reports **36**

H Certifying an instance edition and a route result **38**

Licence and AI-use statement **40**

1 Purpose, scope and conformance

MAMUT-routing publishes benchmark instances alongside the routes of their reported best-known solutions [Pic+26a]. A researcher who compares a new route with the published one needs to know what a number in the instance means, which constraints the route satisfies and which objective the benchmark ranks. A benchmark edition, i.e. the published and immutable state of an instance variant, fixes those choices, and the program that applies them to a submitted route is called the *checker*. We consider the single-depot Capacitated Vehicle Routing Problem (CVRP) and the Vehicle Routing Problem with Time Windows (VRPTW), together with their Time-Dependent (TD) counterparts. In all of them, vehicles share a common capacity and every customer must be visited exactly once. The standard does not cover split deliveries, multiple depots, heterogeneous vehicles, pickup loads, a travel-time matrix separate from the cost matrix or a maximum route duration independent of the depot window, since each of these features would require a named extension defining its constraints and arithmetic. Each edition declares its objective and units. Whereas a static benchmark assigns a fixed cost to each directed arc, a TD benchmark assigns it an Arrival Time Function (ATF), so that the duration of a route can depend on the time at which it leaves the depot.

This document is the standard that MAMUT-routing implements. It is explanatory first and meant to be read by humans, so it states the conventions at a high level, gives the reasons behind them and works through examples, while it leaves implementation-level details such as file layouts, field names and parameters to the open-source code of MAMUT-routing. By design, the text therefore does not suffice on its own to reimplement a checker. It nevertheless takes precedence over the code. A contradiction between the two should never happen, and if one is found, the code is in error and is corrected to follow this document. This text is version 0.1 of the specification, and the instance manifests and results that follow it carry the specification identifier `mamut-routing/0.1`.

The checker needs the same numerical arc data that define the benchmark, and a family can supply them in several ways. It can embed a cost matrix in its instance file, or keep the arc data in an accompanying file called a sidecar, which for TD routing can hold the knots defining the ATFs.[1] It can also supply compact inputs together with a pinned procedure that reconstructs those values. For example, a road graph can yield a static matrix of shortest-path distances, and a road graph with a traffic overlay can yield a collection of TD arc functions. In either case, a verified reconstruction stands in for the stored JSON. To make this verification possible, the publisher pins the input files and code to specific versions and records a SHA-256 digest of the complete interpreted matrix or set of knots. This digest acts as a fingerprint of the agreed

[1] Visser and Spliet [VS20, Sec. 3.1] and Blauth et al. [Bla+24, Sec. 2] use *breakpoint* for the points defining a continuous piecewise-linear function. We use *knot* for each stored pair of departure and arrival times because the sidecar also preserves collinear points and identical consecutive pairs, and can store distinct arrival times at the same departure time to encode a jump. We deliberately retain every knot so that independent implementations can reproduce and verify the same ordered sequence of binary64 operands.

sequence of numerical values. An independent implementation can then regenerate the values, compare the digest and check routes against the verified data. Section 4 explains what must be pinned and when this verification can be reused.

The profiles `binary64-exact-v1` and `td-binary64-exact-v1` fix the same boundary between materialization and evaluation. Each numerical arc or time operand has the value obtained by reading it in IEEE 754 binary64, and the checker uses that value exactly in all subsequent arithmetic. The static profile sums the costs of the traversed arcs, while the TD profile evaluates the published arc functions and finds, for each route, the earliest depot departure attaining the minimum feasible duration. We distinguish a *profile*, which specifies the answer, from a *backend*, which is an implementation of that profile, for example a Python evaluator using rational numbers or a native evaluator using integers where the arithmetic permits. A result identifies both.

A published profile identifier keeps its meaning independently of later versions of this document and of the library that implements it. Hence, a result stored under `binary64-exact-v1` or `td-binary64-exact-v1` can be replayed under the same rules for as long as its inputs remain available, and a change to any rule of a profile, however small, receives a new identifier instead of redefining the existing one.

The words MUST, MUST NOT, SHOULD and MAY express requirements, prohibitions, recommendations and permitted choices within a profile. An implementation claiming a profile MUST give its specified feasibility verdict and authoritative objective for supported inputs. If it cannot finish, for example because a fixed-width integer would overflow, it MUST report failure or an inconclusive outcome. Note that a finite fixture suite gives evidence about the cases it exercises, but conformance over the full input domain also depends on the algorithm and its implementation.

Three questions arise when two checkers disagree, or when both reproduce a value that surprises the reader.

Operational conformance. Did the checker perform the operations required by the declared profile? A sequential floating-point sum can conform to a released checker's recipe even when rounding after each addition gives a different answer from summing the same operands exactly.

Mathematical fidelity. Which mathematical problem do those operations solve? Summing published road distances exactly gives the exact length of the route in that matrix, yet this sum still inherits the rounding of each distance if the generator applied one. Likewise, exact TD interpolation evaluates the published knots, whose resolution was fixed when the arc function was generated.

Reproducibility. Can another team obtain the same inputs and repeat the evaluation? Its

regenerated arc values can be checked against the published digest before either team evaluates a route. Once the inputs match, the teams can investigate any remaining discrepancy in scheduling, aggregation or display.

2 From JSON numbers to exact operands

The JavaScript Object Notation (JSON) format represents numbers as text. For instance, `0.1` is a numerical token, i.e. a sequence of characters accepted by the JSON number grammar, whereas the quoted value `"0.1"` is a JSON string. A numeric instance field requires the former, although a result format may explicitly require strings for the digits of a large numerator or denominator. A JSON decoder also distinguishes the numerical token `1` from the Boolean literal `true`. Beyond these syntactic distinctions, the grammar alone leaves the numerical interpretation of a token to its consumer. JSON interoperability guidance identifies $[-(2^{53}-1), 2^{53}-1]$ as the range within which binary64 consumers can agree on all integer values [Bra17].

The *role of the field* therefore matters. Structural integers, including vertex indices and route counts, are read as integers, whereas arc costs, time-window bounds, service durations and ATF knot coordinates are numerical operands. Their binary64 reading also applies to tokens written without a decimal point. Let $\mathcal{T}_{\mathrm{JSON}}$ be the set of valid JSON numerical tokens and $\mathrm{dec}(t)$ the exact decimal rational spelled by token $t \in \mathcal{T}_{\mathrm{JSON}}$. The token `0.1` spells $1/10$, as do `0.10` and `1e-1`. We write $\mathbb{Q}$ for the rational numbers, $\mathbb{Z}$ for the integers and $\mathbb{N}_0$ for the nonnegative integers. Appendix A collects the numerical domains, conversion operators and routing symbols.

Reading a binary64 value. IEEE 754 binary64 is the 64-bit double-precision floating-point format, in which a normal finite value has 53 significant binary digits. The spacing between successive representable values changes with their magnitude, and the unit in the last place (ulp) describes this local precision. At input we round the exact decimal value to the nearest binary64 value, with a tie going to the representation whose significand has an even least significant bit. We denote this nearest-even rounding by RN_{64}. We require gradual underflow, including finite subnormal values, and a token whose nearest binary64 value is zero contributes zero. We normalize signed zero to positive zero for numerical comparison, semantic digests and result serialization, while the raw file retains the originally written bytes [Sun].

Let $\mathbb{F}_{64}$ be the set of finite binary64 numerical values and $\overline{\mathbb{F}}_{64} = \mathbb{F}_{64} \cup \{-\infty, +\infty\}$. Every $f \in \mathbb{F}_{64}$ is a dyadic rational, i.e. a number of the form $\mu 2^{-b}$ with $\mu \in \mathbb{Z}$ and $b \in \mathbb{N}_0$, and we write $\iota_{64} : \mathbb{F}_{64} \hookrightarrow \mathbb{Q}$ for the inclusion that expresses f as a rational while preserving its value. Reading a numerical token for either exact profile thus takes two steps, the first of which performs the rounding.

$$f = \mathrm{RN}_{64}(\mathrm{dec}(t)) \in \mathbb{F}_{64}, \qquad x = \iota_{64}(f) \in \mathbb{Q}. \tag{1}$$

For `0.1`, the resulting rational is 3602879701896397/36028797018963968. In Python, `Fraction.from_float(float("0.1"))` returns this value, whereas the call `Fraction("0.1")` asks Python to parse the decimal characters directly and therefore returns 1/10. Both fractions are exact for their respective arguments [Pyta; Pytb].

Algorithm 1 Read an arc or temporal operand for either exact profile

Require: An unquoted JSON numerical token $t \in \mathcal{T}_{\mathrm{JSON}}$

1: $f \leftarrow \mathrm{RN}_{64}(\mathrm{dec}(t))$
2: Reject if $f \notin \mathbb{F}_{64}$. Normalize its signed-zero encoding.
3: $x \leftarrow \iota_{64}(f)$
4: Check the numerical domain required by the field.
5: **return** x

A loader MUST distinguish a token's role before a conversion can lose information. It MUST reject duplicate JSON object members, a Boolean or string in a numerical operand field, the non-JSON tokens `NaN` and `Infinity`, and numerical tokens whose binary64 conversion overflows. A parser can use the original token text or an equivalent correctly rounded conversion, while structural integer fields retain their integer syntax and domain. For instance, the Python reference uses token-preserving hooks in the JSON decoder [Pytc].

Printed decimals and decimal grids. A shortest round-trip serialization writes enough decimal digits for a correctly rounded reader to recover the same binary64 value, whereas the exact decimal expansion of that value would often take many more digits. Different spellings can therefore change a file's byte hash while preserving every interpreted operand. A printed number with 3 fractional digits can also come from a generator that quantizes distances to the grid $\mathbb{G}_3 = 10^{-3}\mathbb{Z}$, on which the exact decimal token `215.359` denotes 215359/1000. Since its reduced denominator contains a factor of 5, the nearest binary64 value differs from the grid value, and the binary64 profiles use that rounded value even when every published token was generated to 3 decimal places. Multiplying such a binary64 operand by 1000 and converting it to an integer can therefore change the instance that a solver evaluates.

Rounding during an evaluation. For two binary64 operands $f, f' \in \mathbb{F}_{64}$, a rounded addition means

$$f \oplus_{64} f' = \mathrm{RN}_{64}\Big(\iota_{64}(f) + \iota_{64}(f')\Big) \in \overline{\mathbb{F}}_{64}. \tag{2}$$

The addition within RN_{64} is exact, while repeating $\oplus_{64}$ rounds after each operation. For example, if $f = 2^{53}$, then $(f \oplus_{64} 1) \oplus_{64} 1 = f$, whereas $f \oplus_{64} (1 \oplus_{64} 1) = f + 2$. This failure of associativity explains why the summation order matters for a released profile that specifies rounded additions, since reassociation or wider unrounded intermediates can change such a recipe. In the exact profiles, by contrast, ordinary $+$ denotes rational addition and the exact sum in this example is $a + 2$. An exact static sum can, however, exceed the binary64 range, and TD interpolation

can introduce denominators with factors other than 2, so Section 6 specifies how to store and display the resulting rational objectives.

3 Four contracts for a benchmark result

A result must identify both the problem that was solved and the meaning of its reported value. The following four contracts separate the decisions involved.

Contract	Question	Example obligation
Generation	How did the benchmark authors obtain the published data?	Identify the source graph, routing metric and rounding used to produce the distances.
Instance	Which problem and numerical values does this edition define?	Declare units, constraints and the default profile. Pin the materialization of the matrix or ATFs.
Evaluation	What does a submitted route achieve on those values?	Check all visits, capacities and windows. For TD duration, select the earliest minimizing departure.
Result	What is ranked, and what does the claim establish?	Store the reduced rational objective with the profile and the backend used for certification.

For example, a static family may publish road distances rounded to 3 decimal places. The generation contract then explains where the distances and their rounding came from, the instance contract fixes the binary64 values a reader obtains, and the evaluation contract sums the traversed arc values exactly. Finally, the result contract records that sum and ties it to the verified instance and route set. A different solver may search with binary64, scaled integers, interval bounds or rational arithmetic, yet each submitted incumbent is checked under the same contract for its benchmark edition. A proof of optimality additionally requires a bound for that objective on that feasible set.

4 Identify and materialize the instance

A static checker consumes a matrix c of costs from vertex i to vertex j, whereas a TD checker consumes a common departure horizon and, for each directed arc, an ordered sequence of knots of its ATF. We call the process of obtaining these values from the published instance

materialization. Once the binary64 values are fixed, a checker reading an entire JSON sidecar and a checker regenerating values on demand face the same evaluation problem.

Identify the published inputs. The *instance manifest* is the record that tells a consumer which files and numerical rules belong to this edition. It MUST identify the family and edition, problem type, objective, units, default profile, supported features, and the descriptor and required sidecars by path and SHA-256. The descriptor supplies the depot and customer indices, demands, capacity, fleet limit and applicable temporal data. A result refers to its manifest by the SHA-256 of its file bytes. Appendix H gives the publication order and explains how to verify an edition outside a Git checkout. The customer count determines the expected matrix dimensions or directed arc set. In this core, demands and vehicle capacity are nonnegative integers exactly representable in binary64, and the depot has zero demand and zero service duration. A supplied fleet limit is a nonnegative integer, and an absent limit imposes no bound. Since changing a time window can change feasibility while preserving all arc values, the result identifies the complete manifest.

For a static instance, `units.cost` names the unit of the arc matrix. In a VRPTW the same matrix also supplies the travel times, so `units.time` uses the same unit for windows and service durations. In a CVRP the checker uses the matrix only for the objective. For example, the Mamut2026 fastest and shortest CVRP variants measure travel time in seconds and distance in metres, respectively. Both have the same capacity constraints.

A family descriptor resolves its relative sidecar paths from the family collection root, which is marked by `mamut-collection.json`. For example, the Rifki-30 descriptor in `TDVRP/n=30/` refers to `sidecars/n=30/Rifki-30.atf.json.gz` under the Rifki2020 root. A standalone example carrying its own sidecar instead resolves the path from its descriptor's directory. For a compressed artifact, the manifest MUST specify whether a digest covers the distributed compressed bytes or the decompressed content, because recompressing identical content can change the former digest. The published collection resolver documents the path rules.

Choose the source of arc values. The manifest's `materialization` object identifies the representation and its `interpretation`, here `binary64`. A static instance can use an `embedded-matrix` or a `sidecar-matrix`, and a TD instance can supply a `stored-atf` sidecar. In these cases, the checker reads the published entries or knots directly. A family using `regenerated` data MUST identify the raw inputs by content hash, the materialization function and immutable implementation, the numerical parameters and the relevant execution environment. Record the callable, identify its preserved source distribution by content hash and pin the dependencies that affect numerical output, including parameters for rounding, path selection or function simplification where used. A versioned environment record may be referenced by hash across many manifests.

For example, a Mamut2026 base can yield both a shortest-distance matrix and a fastest-travel-time matrix, and path selection as well as the subsequent 3-decimal rounding are part of the respective materialization recipes. For a TD road model, the traffic overlay can change the arrival function on a path even when the road graph and vertices stay fixed. Other TD families derive their ATFs from speed periods and arc categories, or distribute already computed knots. In each case, the fixed operands reaching the checker are the output of the declared recipe. In particular, coordinates included for display do not authorize replacing road costs by Euclidean distances [Pic+26b; Pic+26c].

Verify the complete materialization. A raw SHA-256 checks the identity of a particular file, whereas a *semantic digest* checks the complete sequence of interpreted binary64 operands. For a static matrix, the manifest field is `materialization.matrix_sha256`, which covers the dimension and all entries in row-major order, including the diagonal. For TD data, the field is `materialization.atf_stream_sha256`, which covers the vertex count, horizon and ordered knots on every directed arc between distinct vertices. Preserving the knot order matters at a jump, where the first and last ordinates at one time have different roles. In released TD descriptors, the separate field `td.atf_sha256` covers the canonical ATF JSON content, including its metadata. Appendix B gives the exact byte streams for the two semantic digests and companion examples.

A verifier MUST compare the complete interpreted stream with the declared semantic digest, although it may produce rows or arcs incrementally and discard each after hashing. A checker may reuse a successful full verification while the raw inputs, code, environment, parameters, interpretation and expected digest remain unchanged. A checker evaluating one submitted solution can then request just its traversed arcs, provided that the on-demand call produces the corresponding entries of the verified materialization. Note that hashing that subset alone only establishes which operands the check used, whereas checking the identity of the whole instance requires the complete stream. A missing required input, digest mismatch or unavailable required materializer stops certification.

Historical generation documentation records how the publisher obtained the descriptor, raw inputs or stored sidecars, including the generator revision and parameters when known. An imported dataset may have a precisely identified source file even when part of its earlier construction is unknown. By contrast, the information needed to read or regenerate the authoritative arc values belongs to the instance contract, because every checker must be able to recover those values.

5 Feasibility and cost of a submitted solution

A submission specifies which customers each vehicle visits and in what order. From these sequences we must decide whether the vehicles can serve all customers and, if so, what the declared objective is worth. The two questions already differ on a route with constant travel times. A vehicle can wait for a customer to open without changing the travel cost of its tour, although that wait does increase the time the vehicle spends away from the depot. We distinguish these quantities throughout, since static benchmarks charge for the arcs traversed, whereas time-dependent benchmarks charge for the elapsed duration of the route.

Both profiles start from the instance identified in Section 4, and under `binary64-exact-v1` (static) and `td-binary64-exact-v1` (TD) each finite binary64 operand f contributes its exact rational value $\iota_{64}(f)$. In particular, a service start exactly at its deadline satisfies that constraint, whereas any strictly later start violates it, however small the excess. The same interpretation applies to arc values, the horizon, window endpoints, service durations and any fixed vehicle charge. Appendix A collects the notation.

5.1 Which customers does each vehicle serve?

Let $\mathcal{C} = \{1, \dots, n\}$ be the customer set and let vertex 0 be the depot. A route is a vector $\mathbf{r} = \langle v_1 = 0, v_2, \dots, v_{m-1}, v_m = 0 \rangle$ of length $m \geq 3$, whose JSON array contains just the nonempty customer subsequence $\mathbf{r}_{\mathcal{C}} = \langle v_2, \dots, v_{m-1} \rangle$. For example, the array `[2,1]` prescribes the three directed arcs $(0, 2)$, $(2, 1)$ and $(1, 0)$. Since these arcs are directed, reversing the array can change both cost and feasibility.

Before checking visits, we validate the instance. The static matrix has one row and column per vertex, including the depot, with $c_{ij} \geq 0$ and $c_{ii} = 0$. Customer demands q_i and vehicle capacity Q are nonnegative integers exactly representable in binary64, and the depot demand is $q_0 = 0$. Each supplied service duration satisfies $s_i \geq 0$, and each window satisfies $0 \leq e_i \leq l_i$. The checker must reject a nonzero depot service duration, even on a route whose customers have no windows. In the TD model all arcs share the finite departure horizon $0 \leq h_0 < h_1$, although windows and the route itself may restrict the feasible departures to a single instant within that horizon.

We first read the submitted enumeration $\mathbf{s}$ with all entries present, in which every entry in a customer subsequence must be an index in $\mathcal{C}$ and every customer must occur exactly once. Checking this on the enumeration before forming a set detects, for example, a file that submits the same route twice. For every route we then require $\sum_{j=2}^{m-1} q_{v_j} \leq Q$, and empty customer sequences are invalid. If a fleet limit K is supplied, the number of routes must be at most K. Note that with $n = 0$ the empty solution is admissible and its cost is zero.

Once these checks succeed, $\mathcal{S}$ denotes the set of routes and $|\mathcal{S}|$ is the realized fleet size. For storage we sort the routes lexicographically by their customer subsequences, which gives canon($\mathcal{S}$) without changing any visit order within a route. The exact departure and duration of each TD route accompany that route in this stored order (Section 6).

5.2 From depot departure to depot return

Consider a vehicle that leaves vertex i at time d_i. In the static problem with time windows, the matrix entry c_{ij} is both the cost and the travel time of the directed arc (i, j). In the TD problem, the ATF α_{ij} instead gives the arrival clock time at j for a departure at i. The common forward calculation at a customer j is

$$u_j = \begin{cases} d_i + c_{ij} & \text{static,} \\ \alpha_{ij}(d_i) & \text{TD,} \end{cases} \qquad \text{(arrival)}$$

$$t_j = \begin{cases} \max(u_j, e_j) & \text{with window } [e_j, l_j], \\ u_j & \text{without a window,} \end{cases} \qquad \text{(service start)}$$

$$d_j = t_j + s_j. \qquad \text{(departure)}$$

The window constrains the start of service, so we require $t_j \leq l_j$ whenever the window is present, while service may finish after l_j. On the last arc we use the same arrival rule with destination 0 and check the depot return deadline l_0 when specified. The elapsed duration ends when the vehicle reaches the depot. For static capacitated routing without time windows, by contrast, the coverage, capacity and fleet checks suffice for feasibility.

In the static problem with time windows we can start at depot opening, $d_0 = e_0$, because later departures can only delay arrival on a constant-time arc. At a customer, both waiting until opening and adding service preserve that order, so induction along the route shows that this schedule gives the earliest attainable service start at each customer and the earliest return to the depot. Hence, if it misses a deadline, every schedule for the same order misses that deadline. Note that starting the clock at the declared opening matters when $e_0 > 0$.

For example, suppose both legs of a one-customer route take 2 time units, the customer accepts service starts in $[7, 9]$, service takes 2, and the depot window is $[0, 15]$. A vehicle leaving at 0 arrives at 2, waits until 7, finishes service at 9 and returns at 11, so the route is feasible. Its static travel cost is $2 + 2 = 4$, yet the vehicle has spent 11 time units away from the depot. We now keep the same customer sequence and allow the departure time to vary.

5.3 Choosing the depot departure in the TD problem

A TD instance gives every directed arc a finite sequence of knots (x_k, y_k) spanning a common closed departure horizon $[h_0, h_1]$, where each knot relates a departure clock time to an arrival clock time. Since the abscissas and ordinates are nondecreasing and $y_k \geq x_k$, traversal takes nonnegative time and the First-In, First-Out (FIFO) property holds on each arc. In other words, if two vehicles enter the same arc in succession, the second can reach its head no earlier than the first.

Interpolation and jumps. Let $y^-(x)$ and $y^+(x)$ be the first and last ordinates stored at a distinct abscissa x. At x itself the checker uses $\alpha_{ij}(x) = y^-(x)$, including at h_0 and h_1, whereas for consecutive distinct abscissas $x < x'$ and an interior departure $x < t < x'$ it interpolates by

$$\alpha_{ij}(t) = y^+(x) + \frac{t - x}{x' - x}\Big(y^-(x') - y^+(x)\Big). \tag{3}$$

When several knots have the same abscissa, the lower ordinate is attained at that instant, and if another piece follows, it starts from the upper ordinate on the right. This distinction matters for a vehicle that can meet a deadline by leaving exactly at the jump, and Appendix F works through such a departure on a published instance. Each departure onto an arc, including the return leg, must lie in $[h_0, h_1]$, but its arrival may exceed h_1 on the final leg if the depot deadline permits it or if no deadline applies.

Minimum elapsed time. Choose an initial depot departure t from the continuous horizon intersected with the depot window when present, and apply the forward calculation of Section 5. For this fixed t, starting service as soon as the customer allows gives the earliest possible return. Indeed, a voluntary delay at a customer can only move the next departure later, and FIFO preserves that order along every remaining arc. What can change the elapsed time is the choice of t itself.

Let $D_{\mathbf{r}}$ be the set of initial departures for which the forward schedule meets all windows and arc horizons, and write $\delta_{\mathbf{r}}(t)$ for its depot return time. The route is infeasible if $D_{\mathbf{r}}$ is empty, and otherwise the route evaluation returns both the minimum duration and the earliest departure attaining it,

$$\Delta^*_{\mathbf{r}} = \min_{t \in D_{\mathbf{r}}}\Big(\delta_{\mathbf{r}}(t) - t\Big), \tag{4}$$

$$t^*_{\mathbf{r}} = \min\{t \in D_{\mathbf{r}} \mid \delta_{\mathbf{r}}(t) - t = \Delta^*_{\mathbf{r}}\}. \tag{5}$$

Note that the departure ranges over real times even when a family expresses time in integer milliseconds. Under the knot convention above, both minima are attained, including when the

feasible set consists of a single departure. Moreover, the earliest minimizing departure and the corresponding duration are rational because the stored knots, window endpoints and service durations are rational (Appendix E).

For the one-customer route above, take $[h_0, h_1] = [0, 15]$ and give each leg the ATF $t \mapsto t + 2$. Departures in $[0, 5]$ all return at 11 because the customer opens at 7, whereas each later departure through 7 avoids waiting and returns at $t + 6$. Departing after 7 misses the customer's latest start. Hence the minimum duration is 6, attained throughout $[5, 7]$, and the route record gives departure 5. If the depot instead closes at 10, the earliest possible return of 11 makes the route infeasible under either profile.

5.4 From route values to the solution objective

A solution receives an authoritative cost only when all its routes are feasible. The benchmark manifest declares which objective to minimize. For a static route $\mathbf{r}$, let $c_{\mathbf{r}} = \sum_{j=1}^{m-1} c_{v_j v_{j+1}}$, including both depot legs. With $C(\mathcal{S}) = \sum_{\mathbf{r}\in\mathcal{S}} c_{\mathbf{r}}$, `MonoCost` minimizes $C(\mathcal{S})$, whereas `HierarchicalVehicleCost` minimizes the pair $(|\mathcal{S}|, C(\mathcal{S}))$ lexicographically. Under the latter objective, any feasible solution using fewer vehicles ranks first, whatever its travel cost.

Because the vehicles of a TD solution choose their departures independently, the objective `Duration` sums the exact minima $\sum_{\mathbf{r}\in\mathcal{S}} \Delta^*_{\mathbf{r}}$. For `FleetCostDuration` the instance also supplies a nonnegative fixed charge F per used vehicle, in the same units as duration, and the scalar objective becomes $F|\mathcal{S}| + \sum_{\mathbf{r}\in\mathcal{S}} \Delta^*_{\mathbf{r}}$. In both cases the elapsed duration includes travel, customer service and waiting away from the depot. Since all four objectives use exact sums, route enumeration has no effect on their values. Section 6 specifies how to store and compare the resulting exact keys, and Table 2 summarizes the 4 objectives.

When a displayed departure misses the deadline. An exact departure serves as a witness because we can replay the route at that instant, whereas its nearest binary64 display value can lie on the other side of a deadline. Consider, for instance, the customer sequence $\langle 2, 1, 4, 9, 8, 10, 3, 6, 5, 7\rangle$ in the 10-customer Blauth-Nairobi instance. Its knots are in integer milliseconds, but the earliest minimizing departure is

$$t^*_{\mathbf{r}} = \frac{46757961125227679073}{790061513395} \text{ ms.}$$

At that departure the vehicle reaches customer 1, its second visit, exactly at the deadline of 61 200 000 ms. The nearest binary64 departure has shortest round-trip spelling `59182684.30049512` ms, and replaying that binary64 value with the same exact ATFs misses the deadline by approximately 9.73×10^{-10} ms. The rational witness, in contrast, reproduces

the feasible schedule and its objective even at this boundary. The materialized route and its exact evaluation are identified in the archived route panel.

6 From a checked route to a stored result

Suppose two teams submit routes for the same instance. To decide whether either team has improved the Best-Known Solution (BKS), the curator needs the routes, the edition of the instance and its objective. Under both default profiles these suffice to recompute feasibility and the objective key, and a submission can therefore consist of routes and an instance identity. A stored BKS then records the outcome of checking that submission, with enough information for another reader to identify the same numerical problem, reproduce the result and compare a new candidate.

6.1 The exact value and its display

Consider the static example distributed with this specification, whose route visits customer 1, then customer 2, and then returns to the depot. The three arc costs in the instance JSON are `0.1`, `0.2` and `0.3`, and after binary64 interpretation their exact sum is

$$C = \frac{21617278211378381}{36028797018963968}.$$

The nearest binary64 value has the round-trip spelling `0.6`. A reader who reconstructs a fraction from that binary64 value obtains a rational slightly smaller than C, whereas a reader who interprets the spelling directly as decimal obtains $3/5$. Neither reading gives the actual cost, which we recover by reading the fraction or by evaluating the three arcs under the declared profile. In `example-result.json`, the top-level `cost` is `0.6`, while inside `metadata.standard` the same result has

```
"authoritative_cost": {
  "numerator": "21617278211378381",
  "denominator": "36028797018963968"
}
```

Both profiles encode an exact value p/q in lowest terms with $q > 0$ and write both integers as base-10 strings, without a leading plus, redundant leading zeros or negative zero. Encode zero as numerator `"0"` and denominator `"1"`. A reader MUST reject a noncanonical fraction. We use strings because they permit clients that read JSON numbers as binary64 to retain arbitrarily large integers without loss [Bra17].

For the top-level `cost`, round the exact total once to nearest binary64 with ties to even, and emit a finite projection as a round-trip JSON number, with positive zero. If that projection overflows, emit `cost:null`, set `display_overflow` to true and retain the exact fraction. A website showing d decimal places instead rounds the exact total directly to the nearest point on $10^{-d}\mathbb{Z}$, again with ties to even. Converting to binary64 first and rounding again can produce a different last digit through double rounding.

6.2 What the TD witness adds

In the companion TD instance the vehicle capacity is 1 and each of the two customers has demand 1, so the solution uses the two routes `[1]` and `[2]`. On the first route, the outward arc gives arrival $1 + 3t/2$ at departure t, customer 1 opens at 6 and needs 1 unit of service, and the return trip takes 1. Leaving before $10/3$ adds waiting, whereas leaving after $10/3$ increases elapsed duration because the outward arrival grows faster than the departure. At $10/3$ the vehicle reaches the customer exactly at opening, returns at 8 and spends $14/3$ time units away from the depot. On the second route, any departure in $[5, 7]$ attains duration 6 and the earliest of them is 5. With a fixed charge of $5/4$ per vehicle, `FleetCostDuration` therefore gives

$$C = \frac{14}{3} + 6 + 2\left(\frac{5}{4}\right) = \frac{79}{6}.$$

A TD BKS MUST store `metadata.standard.route_evaluations` in the same order as its stored `routes`. Every entry contains both the exact `duration` and the exact earliest `departure_time`, as in the following array for the two routes above.

```
[
  {"duration":       {"numerator":"14", "denominator":"3"},
   "departure_time": {"numerator":"10", "denominator":"3"}},
  {"duration":       {"numerator":"6",  "denominator":"1"},
   "departure_time": {"numerator":"5",  "denominator":"1"}}
]
```

Replaying the first departure checks that the vehicle attains $14/3$, while independently minimizing the duration also checks that this value is optimal and that no earlier feasible departure attains it. A displayed departure may be rounded to binary64 for a user interface, but a client that replays a schedule uses the exact fraction.

6.3 Record identity and compare like with like

In both examples, `routes` holds the customer sequences in lexicographic order, with depot endpoints implicit. Preserve customer order within a route, and sort routes and their associated TD evaluations together. Table 1 describes the contents of `metadata.standard`, and the complete files `example-result.json` and `example-td-result.json` include real hashes of their companion manifests and of the checkers used to produce them.

Field	What a reader verifies
`specification`	The specification version used to produce the result, here `mamut-routing/0.1`.
`benchmark_revision`	The edition label of this instance variant within its family, for instance `r1`. The field keeps its historical name and holds this label.
`manifest_sha256`	SHA-256 of the complete manifest file bytes. The manifest binds the descriptor and the materialized arc data.
`profile`, `objective`	The arithmetic contract and the mathematical objective declared for this edition.
`num_routes`, `routes_sha256`	The number of vehicles used and the digest of the canonical customer sequences.
`authoritative_cost`	The reduced fraction giving the exact scalar cost component.
`display_overflow`	True exactly when binary64 projection of the total overflows.
`evaluation`	The actual backend, checker identity, immutable revision or build digest, runtime identity and verification status. Automatic selection records the backend used.
`route_evaluations` (TD)	The exact duration and earliest departure of each route in stored order.

Table 1: Fields shared by the static and TD results, with the additional route witnesses required for TD.

To reproduce `routes_sha256`, hash an ASCII stream beginning with `MAMUT-ROUTES-v1` and a line feed (LF), followed by the route count and LF. Each stored route then contributes its customer count, a colon, its comma-separated customer indices and LF. Use base-10 integers without leading zeros. For the static example, the three successive lines are `MAMUT-ROUTES-v1`, `1` and `2:1,2`, including the final LF.

Since a supplied cost or schedule is an assertion, recompute the exact values and reject an inconsistent result, including a feasible TD witness that is later than the earliest minimizing departure.

The backend record lets another team repeat the evaluation in the same implementation or check it independently. Python `Fraction` supplies a selectable reference for each profile, whereas the native defaults of the library use prepared C++ dyadic arithmetic for static evaluation and C++ prefix composition with the GNU Multiple Precision Arithmetic Library (GMP) for TD. Appendix C explains how the algorithms obtain equal exact outputs despite their different representations of numbers and schedules.

6.4 Rank a candidate against the incumbent

Each objective ranks solutions by an exact comparison key, which Table 2 gives together with the definitions of Section 5.3, the unit of the cost and the families whose manifests declare it. Keys are compared lexicographically, so the hierarchical objective compares vehicle counts before travel costs, whereas `FleetCostDuration` already prices each vehicle in its scalar cost and thus has a single component. An incumbent is replaced only when the candidate strictly improves the complete exact key, and on equality the incumbent already in the store is kept. For example, scalar costs 1 and $1 + 2^{-54}$ both display as binary64 1, yet the first is strictly smaller.

Objective	Minimized key	Unit of cost	Problem types and families
`MonoCost`	$(C(\mathcal{S}))$	arc-cost unit	CVRP of Mamut2026 and Poryos2026 (metre or second), VRPTW of Dimacs2021 (one tenth of the Solomon unit), Ortec2022 and Poryos2026 (second)
`HierarchicalVehicleCost`	$(\lvert\mathcal{S}\rvert, C(\mathcal{S}))$	vehicles, then arc-cost unit	VRPTW of Sintef2008 (Solomon unit)
`Duration`	$(\sum_{\mathbf{r}\in\mathcal{S}} \Delta^*_{\mathbf{r}})$	time unit	TDVRP and TDVRPTW of Ari2018, Dabia2013, Lera2026, Vu2020 (unit of the original benchmark), Poryos2026 and Rifki2020 (second)
`FleetCostDuration`	$(F\lvert\mathcal{S}\rvert + \sum_{\mathbf{r}\in\mathcal{S}} \Delta^*_{\mathbf{r}})$	time unit, with F in the same unit	TDVRPTW of Blauth2024 (millisecond)

Table 2: Objectives and their exact comparison keys. Here $C(\mathcal{S})$ is the exact sum of the traversed arc costs, $\Delta^*_{\mathbf{r}}$ the minimum duration of route $\mathbf{r}$ and F the fixed charge per used vehicle. The last column lists the families whose edition manifests declare the objective, with the cost unit they declare.

Compare a candidate with the incumbent only within the same benchmark edition, interpreted inputs, objective and profile. A stored cost, like a submitted one, is an assertion, so both route sets MUST be re-evaluated under the profile of the family before their keys are compared. If the stored incumbent turns out to be infeasible under that profile, the publisher can fall back on the earlier route sets recorded for the same inputs and objective, the best feasible one then taking its place in the comparison. If none of them is feasible, the instance has no best-known solution for that objective until a feasible candidate is checked.

References

[Pic+26a] Adrien Pichon, Florian Rascoussier, Romain Billot, Lina Fahed, Alexandru-Liviu Olteanu, Marc Sevaux, and Christine Solnon. *MAMUT-routing*. Static source snap-

shot used for the inventory and numerical comparisons. Author order and ORCIDs from CITATION.cff. 2026. URL: https://github.com/ANR-MAMUT/MAMUT-routing-archive-2026-11/tree/eca336ff4b087f5c2f7b9ac7c2ab33668ff7af2b.

[VS20] Thomas R. Visser and Remy Spliet. "Efficient Move Evaluations for Time-Dependent Vehicle Routing Problems". In: *Transportation Science* 54.4 (2020), pp. 1091–1112. DOI: 10.1287/trsc.2019.0938.

[Bla+24] Jannis Blauth, Stephan Held, Dirk Müller, Niklas Schlomberg, Vera Traub, Thorben Tröbst, and Jens Vygen. "Vehicle routing with time-dependent travel times: Theory, practice, and benchmarks". In: *Discrete Optimization* 53 (2024), p. 100848. DOI: 10.1016/j.disopt.2024.100848.

[Bra17] Tim Bray. *The JavaScript Object Notation (JSON) Data Interchange Format*. RFC 8259. Dec. 2017. DOI: 10.17487/RFC8259. URL: https://www.rfc-editor.org/rfc/rfc8259.

[Sun] Sun Microsystems. *IEEE Arithmetic*. Numerical Computation Guide, Chapter 2. Historical documentation of the double format and rounding model. URL: https://docs.oracle.com/cd/E19957-01/806-3568/ncg_math.html (visited on 10/23/2026).

[Pyta] Python Software Foundation. *Floating-Point Arithmetic: Issues and Limitations*. Python 3.13 documentation. URL: https://docs.python.org/3.13/tutorial/floatingpoint.html (visited on 10/23/2026).

[Pytb] Python Software Foundation. *fractions: Rational Numbers*. Python 3.13 documentation. URL: https://docs.python.org/3.13/library/fractions.html (visited on 10/23/2026).

[Pytc] Python Software Foundation. *json: JSON Encoder and Decoder*. Python 3.13 documentation. URL: https://docs.python.org/3.13/library/json.html (visited on 10/23/2026).

[Pic+26b] Adrien Pichon, Florian Rascoussier, Romain Billot, Lina Fahed, Alexandru-Liviu Olteanu, Marc Sevaux, and Christine Solnon. *MAMUT-routing Mamut2026*. Family snapshot used for the static inventory. 2026. URL: https://github.com/ANR-MAMUT/MAMUT-routing-Mamut2026/tree/fb339e16475b541cddc3a85bf964c23bebfd543d.

[Pic+26c] Adrien Pichon, Florian Rascoussier, Romain Billot, Lina Fahed, Alexandru-Liviu Olteanu, Marc Sevaux, and Christine Solnon. *MAMUT-routing Poryos2026*. Family snapshot used for the static inventory. 2026. URL: https://github.com/ANR-MAMUT/MAMUT-routing-Poryos2026/tree/94dacfb11d11242280e0bc6f0f8aab300bd9df4a.

[Pytd] Python Software Foundation. *Built-in Functions: round*. Python 3.13 documentation. URL: https://docs.python.org/3.13/library/functions.html#round (visited on 10/23/2026).

A Mathematical notation and arithmetic domains

This appendix defines the number sets, operators and routing symbols used in the prose and pseudocode. Its definitions distinguish the numerical value of an operand from the arithmetic representation used to evaluate it, and domain annotations such as $f \in \mathbb{F}_{64}$ and $x \in \mathbb{Q}$ make that choice explicit without prescribing a particular software type.

For routing objects, sets use calligraphic capitals and ordered vectors use bold lowercase letters. For example, $\mathcal{V}$ and $\mathcal{C}$ are the vertex and customer sets, $\mathcal{S}$ is a solution set, and $\mathbf{r}$ is a route vector. Angle brackets denote ordered vertex sequences and arcs. The ordered route enumeration $\mathbf{s}$ records how a solution is submitted, which matters for an order-sensitive evaluator. Both endpoints of $\mathbf{r}$ use the stored depot index 0, whereas the JSON route arrays encode only $\mathbf{r}_{\mathcal{C}}$. The symbols for vertex and customer sets, routes and solutions, demands and capacity, fleet limit, time windows, service durations, arc costs, arrival functions and route durations follow the routing notation that we share with our other work on these problems. By contrast, the depot index 0, the departure horizon $[h_0, h_1]$, the enumeration $\mathbf{s}$, the schedule and cost symbols u_j, d_j, C and F, the number sets, the parsing and rounding operators, and the variables local to an algorithm or to Appendix C belong to this document alone.

A.1 Number sets and their relationships

We use $\mathbb{N}_0 = \{0, 1, 2, \ldots\}$, $\mathbb{Z}$, $\mathbb{Q} = \{p/q : p \in \mathbb{Z},\ q \in \mathbb{Z}_{>0}\}$ and $\mathbb{R}$ for nonnegative integers, integers, rationals and reals, respectively. For a number set X, the notation $X_{\geq 0} = \{x \in X : x \geq 0\}$ and $X_{>0} = \{x \in X : x > 0\}$ restricts its sign. We further define

$$\mathbb{D} = \{\mu 2^{-b} : \mu \in \mathbb{Z},\ b \in \mathbb{N}_0\}, \tag{6}$$

$$\mathbb{D}_{10} = \{\mu 10^{-d} : \mu \in \mathbb{Z},\ d \in \mathbb{N}_0\}, \tag{7}$$

$$\mathbb{G}_d = 10^{-d}\mathbb{Z} = \{\mu 10^{-d} : \mu \in \mathbb{Z}\}, \qquad d \in \mathbb{N}_0. \tag{8}$$

Thus $\mathbb{D}$ contains every dyadic rational, $\mathbb{D}_{10}$ contains every rational with a finite decimal expansion, and $\mathbb{G}_d$ fixes one decimal grid. For example, $215359/1000 \in \mathbb{G}_3$, while the binary64 value read from `215.359` lies outside that grid.

The IEEE 754 standard defines binary64, a double-precision binary floating-point format whose values occupy 64 bits, with 1 sign bit, 11 exponent bits and 52 explicit fraction bits. For a normal finite value, let $\sigma \in \{0, 1\}$ be its sign bit, $E \in \{1, \ldots, 2046\}$ its exponent field and $T \in \{0, \ldots, 2^{52} - 1\}$ its fraction field, which IEEE 754 calls the trailing significand field. Its exact numerical value is

$$(-1)^{\sigma}(1 + T2^{-52})2^{E-1023}, \qquad 1 \leq E \leq 2046. \tag{9}$$

For $E = 0$, the finite value is $(-1)^\sigma T 2^{-1074}$, which covers subnormal numbers and signed zero, while the all-ones exponent encodes infinities and NaNs. The normal significand has 53 bits of precision [Sun].

The finite binary64 numerical values form the set

$$\begin{aligned}\mathbb{F}_{64} = \{0\} &\\ \cup \{\pm\mu 2^{-1074} : \mu \in \mathbb{Z},\ 1 \le \mu < 2^{52}\} &\\ \cup \{\pm\mu 2^{\eta} : \mu \in \mathbb{Z},\ 2^{52} \le \mu < 2^{53},\ \eta \in \{-1074, \dots, 971\}\}.&\end{aligned} \tag{10}$$

The second set gives the nonzero subnormal values and the third gives the normal values. Since this is a set of numerical values, positive and negative zero both denote its element 0. NaNs and infinities are excluded. The domains satisfy

$$\mathbb{F}_{64} \subsetneq \mathbb{D} \subsetneq \mathbb{D}_{10} \subsetneq \mathbb{Q} \subsetneq \mathbb{R}. \tag{11}$$

For each $d \in \mathbb{N}_0$, $\mathbb{G}_d \subseteq \mathbb{D}_{10}$. Note, however, that a 3-decimal grid value need not belong to $\mathbb{F}_{64}$, and a value in $\mathbb{F}_{64}$ need not lie on the 3-decimal grid. Likewise, an exact sum of binary64 operands belongs to $\mathbb{D}$ and can lie outside $\mathbb{F}_{64}$, while a quotient of differences of such operands can lie outside $\mathbb{D}$. For example, interpolating the arrival between knots $(0, 1)$ and $(3, 5)$ at departure 1 gives $1 + (5 - 1)/3 = 7/3$. The domains of the TD route arithmetic therefore include general rational numbers.

A.2 Parsing, value preservation and rounding

The set $\mathcal{T}_{\text{JSON}}$ contains the character sequences permitted as JSON numbers, and the exact decimal parser $\text{dec} : \mathcal{T}_{\text{JSON}} \to \mathbb{D}_{10}$ interprets the token before any binary64 conversion. The map $\iota_{64} : \mathbb{F}_{64} \hookrightarrow \mathbb{Q}$ is the value-preserving inclusion into exact rational arithmetic, in the sense that, for $f \in \mathbb{F}_{64}$, $\iota_{64}(f)$ has precisely the same numerical value as f. Constructing an exact rational from a binary64 operand thus changes its arithmetic representation without reversing an earlier rounding.

For example, Algorithm 1 applies the following path to the numerical token $t = \texttt{0.1} \in \mathcal{T}_{\text{JSON}}$ under either exact binary64 profile.

$$\text{dec}(t) = \frac{1}{10} \in \mathbb{D}_{10}, \tag{12}$$

$$f = \text{RN}_{64}\big(\text{dec}(t)\big) \in \mathbb{F}_{64}, \tag{13}$$

$$x = \iota_{64}(f) = \frac{3602879701896397}{36028797018963968} \in \mathbb{D} \subseteq \mathbb{Q}. \tag{14}$$

Here x equals the numerical value stored in f, while $x \neq \text{dec}(t)$. Subsequent $+$ operations on x

are exact rational additions, whereas subsequent $\oplus_{64}$ operations on f round after each addition. In Python, the value-preserving conversion is `Fraction.from_float(f)`. Directly applying `Fraction("0.1")` instead implements the decimal interpretation.

The rounding operator $\mathrm{RN}_{64} : \mathbb{R} \to \overline{\mathbb{F}}_{64}$, with $\overline{\mathbb{F}}_{64} = \mathbb{F}_{64} \cup \{-\infty, +\infty\}$, uses the specified binary64 nearest-even rule and IEEE overflow behavior. An input that overflows binary64 is invalid, whereas the exact evaluation may produce an objective larger than every finite binary64 value. In that case, Section 6 preserves its fraction and records overflow of the display projection separately. The operator $a \oplus_{64} b = \mathrm{RN}_{64}(\iota_{64}(a) + \iota_{64}(b))$ is defined for $a, b \in \mathbb{F}_{64}$ and can overflow. Ordinary $+$, $-$ and comparisons in exact algorithms use their mathematical meanings without an implicit rounding step, and ordering and maximum on finite binary64 values use the order inherited from $\mathbb{Q}$.

For $d \in \mathbb{N}_0$, decimal quantization is $Q_d : \mathbb{R} \to \mathbb{G}_d$, defined by $Q_d(x) = 10^{-d}\,\mathrm{roundEven}(10^d x)$, where $\mathrm{roundEven} : \mathbb{R} \to \mathbb{Z}$ selects the nearest integer with ties to even. Algorithm 2 evaluates this map for $x \in \mathbb{Q}$ using integer arithmetic. Note that the symbols Q_d, $\mathbb{Q}$ and the vehicle capacity Q denote a quantization function, a number set and an integer resource bound, respectively.

A.3 Symbols and notation

The table below collects the main symbols, including the quantities local to an algorithm or to Appendix C. We write $\perp$ for an absent mathematical value, corresponding to null in the relevant JSON field. The symbol $\leftarrow$ denotes assignment, $\in$ membership, $\subseteq$ inclusion and $\subsetneq$ proper inclusion.

Table 3: Number sets, operators and routing symbols.

Symbol or notation	Domain and meaning
$\mathbb{N}_0, \mathbb{Z}, \mathbb{Q}, \mathbb{R}$	Nonnegative integers, integers, rational numbers and real numbers.
$X_{\geq 0}, X_{>0}$	Nonnegative or positive elements of a numerical set X.
$\mathbb{D}, \mathbb{D}_{10}$	Dyadic rationals and finite-decimal rationals, respectively.
$\mathbb{G}_d$	Decimal grid $10^{-d}\mathbb{Z}$ for $d \in \mathbb{N}_0$.
$\mathbb{F}_{64}, \overline{\mathbb{F}}_{64}$	Finite binary64 numerical values, and those values extended with infinities. NaNs are in neither set.
$\mathbb{F}_{64,\geq 0}$	Nonnegative finite binary64 values, $\mathbb{F}_{64} \cap \mathbb{R}_{\geq 0}$.
σ, E, T	Binary64 sign bit, biased exponent field and fraction field in Equation 9.
μ, η	Integer coefficient $\mu \in \mathbb{Z}$ and binary exponent $\eta \in \mathbb{Z}$, as in $f = \mu 2^\eta$ for a nonzero $f \in \mathbb{F}_{64}$. In set definitions, μ is the integer coefficient of a power of 2 or 10.
ν, μ', η'	Number ν of trailing zero bits of a positive μ, odd coefficient $\mu' = \mu/2^\nu$ and exponent $\eta' = \eta + \nu$, so that $f = \mu' 2^{\eta'}$.

Symbol or notation	Domain and meaning (continued)
$\mathcal{T}_{\mathrm{JSON}},\ t$	Valid JSON numerical tokens and an individual token $t \in \mathcal{T}_{\mathrm{JSON}}$.
dec	Exact decimal parser $\mathcal{T}_{\mathrm{JSON}} \to \mathbb{D}_{10}$.
f, x	A binary64 operand $f \in \mathbb{F}_{64}$ and an exact rational operand $x \in \mathbb{Q}$, with the chosen interpretation specified.
ι_{64}	Inclusion $\mathbb{F}_{64} \hookrightarrow \mathbb{Q}$ preserving the exact numerical value of a binary64 operand.
RN_{64}	Binary64 nearest-even rounding $\mathbb{R} \to \overline{\mathbb{F}}_{64}$, with overflow checked before acceptance.
$+,\ \oplus_{64}$	Exact addition and addition followed by binary64 rounding, respectively.
$Q_d, \mathrm{roundEven}$	Decimal quantization to $\mathbb{G}_d$ and nearest-even rounding to $\mathbb{Z}$.
P	Numerical evaluation profile of the benchmark edition.
I	Materialized instance, including the applicable matrix or arc functions and the remaining problem data.
$\mathcal{V}, \mathcal{C}, n$	Vertices $\mathcal{V} = \mathcal{C} \cup \{0\}$, customers $\mathcal{C} = \{1, \ldots, n\}$ and customer count $n = \lvert\mathcal{C}\rvert \in \mathbb{N}_0$. Vertex 0 is the depot.
q_i, Q	Demand and vehicle capacity in $\mathbb{N}_0 \cap \mathbb{F}_{64}$, with $q_0 = 0$.
$K, \perp$	Fleet upper bound in $\mathbb{N}_0 \cup \{\perp\}$. The symbol $\perp$ denotes an absent value (JSON null), imposing no fleet limit.
c, c_{ij}	Static arc matrix and its directed entry from i to j. The matrix has shape $(n+1) \times (n+1)$, entries in $\mathbb{F}_{64,\geq 0}$ and $c_{ii} = 0$.
e_i, l_i, s_i	Window endpoints and service duration, with $e_i \leq l_i$ and $s_0 = 0$. Customer windows constrain service start. The depot window bounds departure and return.
$\mathbf{r}, m$	Route vector $\mathbf{r} = \langle v_1 = 0, v_2, \ldots, v_{m-1}, v_m = 0 \rangle$ of length $m = \lvert\mathbf{r}\rvert \geq 3$.
$\mathbf{r}_{\mathcal{C}}$	Nonempty customer subsequence $\langle v_2, \ldots, v_{m-1} \rangle \in \mathcal{C}^{m-2}$, encoded by each JSON route array.
$\mathrm{arcs}(\mathbf{r})$	Ordered arc sequence $\langle \langle v_j, v_{j+1} \rangle \rangle_{j=1}^{m-1}$.
$\mathcal{S}, \lvert\mathcal{S}\rvert$	Set of routes forming a solution, and its realized fleet size. For $n = 0$, $\mathcal{S} = \emptyset$ is permitted.
$\mathbf{s}$	Submitted ordered enumeration of the route vectors, with $\lvert\mathcal{S}\rvert$ entries once validated. Duplicate entries are rejected before forming $\mathcal{S}$.
$\mathrm{canon}(\mathcal{S})$	Lexicographically sorted enumeration of the validated route set, preserving each route's internal customer order.
$c_{\mathbf{r}}, C(\mathcal{S}), C$	Route cost, solution routing cost and current exact accumulator, in $\mathbb{Q}_{\geq 0}$ for exact profiles.
u_j, t_j, d_j	Arrival at customer j, service start and departure after service. When a window is present, $t_j = \max(u_j, e_j)$ and $t_j \leq l_j$. In both settings $d_j = t_j + s_j$.
d_0	Initial depot departure. For the earliest static schedule with a depot window, $d_0 = e_0$. In TD routing d_0 may be chosen from the feasible departure set.

Symbol or notation	Domain and meaning (continued)
$[h_0, h_1]$	Common closed TD arc departure horizon with $0 \leq h_0 < h_1$. Every arc departure lies within it. Route constraints may restrict feasible initial departures to a single time.
$\alpha_{ij}(t)$	Arrival clock time at j after entering directed arc (i, j) at time t. The difference $\alpha_{ij}(t) - t$ is its travel time.
(x_k, y_k)	Stored departure abscissa and arrival ordinate of the kth knot on one ATF. Both are finite binary64 operands.
ϕ_{ij}	Number of materialized knots of the ATF of arc (i, j).
$y^-(x), y^+(x)$	First and last stored ordinates at abscissa x, on one ATF or on a retained prefix function. The first is attained at x. The last is the right-hand limit whenever a following piece exists.
$D_{\mathbf{r}}, \delta_{\mathbf{r}}(t)$	Feasible initial depot departures of route $\mathbf{r}$ and its return clock time for $t \in D_{\mathbf{r}}$.
$\mathbf{r}_j, \delta_{\mathbf{r}_j}$	Prefix $\langle v_1, \ldots, v_j \rangle$ of $\mathbf{r}$ and its ready-time function, i.e. the vehicle clock after v_j as a function of the initial departure, with $\delta_{\mathbf{r}_m} = \delta_{\mathbf{r}}$.
ζ, t_ζ	A time boundary, such as a window bound, horizon end or arc knot, and the initial departure at which a prefix clock reaches it.
ε, γ	Positive displacement of the initial departure in the event sweep, and the nonnegative rate of a clock with respect to it.
$\Delta^*_{\mathbf{r}}, t^*_{\mathbf{r}}$	Minimum elapsed route duration and earliest feasible departure attaining it, defined for a feasible route by Equations 4 and 5.
F	Fixed charge per used vehicle in the TD objective `FleetCostDuration`. Its unit is the instance's duration unit.
$\widehat{C}$	Rounded binary64 projection $\mathrm{RN}_{64}(C) \in \overline{\mathbb{F}}_{64}$. Only a finite value is emitted as numeric compatibility `cost`.
b, d	Binary-scale exponent $b \in \mathbb{N}_0$ and decimal precision $d \in \mathbb{N}_0$, with scales 2^b and 10^d. A tick is the unit 2^{-b} of the common static scale.
$c_{\max}, s_{\max}, e_{\max}, l_{\max}, B$	Largest arc cost, service duration, opening time and closing time of an instance, with zeros for absent temporal fields, and the bound B in ticks on the sums of a static evaluation.
p, q, ℓ, ρ	In Algorithm 2, rational numerator $p \in \mathbb{Z}$, denominator $q \in \mathbb{Z}_{>0}$, division quotient $\ell \in \mathbb{Z}$ and integer remainder $0 \leq \rho < q$. The denominator q is distinct from a customer demand q_i.
N	Number of vertices, $N = n + 1$, in the canonical materialization streams.

B Canonical byte streams for materialized arc data

Two implementations can compare the inputs to their checkers by computing SHA-256 over the complete materialized matrix or collection of ATFs. The manifest records the digest as 64 lowercase hexadecimal digits, in `materialization.matrix_sha256` for a static matrix or in `materialization.atf_stream_sha256` for TD arc data. The instance and manifest hashes identify the remaining data, including units, service durations, windows and vehicle charges. In a published TD descriptor, `td.atf_sha256` additionally identifies the canonical ATF sidecar JSON, including its metadata. A verifier checks each digest against the byte stream named by its field.

Byte conventions. Concatenate the fields in the order given below. Every text field is ASCII, and each line ends with a single LF byte (hexadecimal `0a`). Write each structural integer in base 10, using digits 0 through 9, with a single digit for zero and no leading zeros otherwise. Write each numerical operand as exactly 8 bytes, most significant byte first, containing its IEEE 754 binary64 representation after the instance's binary64 reading. Both signed zeros encode as 8 zero bytes, whereas every other finite value, including subnormals, is preserved. Reject nonfinite operands. Text and binary fields meet directly, without padding, extra separators or a trailing file marker. Hash the bytes themselves, since the hexadecimal text in the companion vectors is a printable representation of those bytes.

Static matrix (`MAMUT-MATRIX-v1`). Let $N = n + 1$ be the number of vertices including depot 0. Write three lines containing `MAMUT-MATRIX-v1`, `binary64` and N, respectively, followed by all N^2 binary64 entries, including the diagonal, in increasing row index $i = 0, \ldots, N-1$ and, within each row, increasing column index $j = 0, \ldots, N-1$. Entry (i, j) is the cost from i to j, and a depot-only instance contributes its single zero diagonal entry.

Directed ATFs (`MAMUT-ATF-v1`). Write three lines containing `MAMUT-ATF-v1`, `binary64` and N, respectively, followed by the 2 binary64 endpoints h_0, h_1 of the common departure horizon, in that order. Next write exactly $N(N-1)$ directed arc records in lexicographic order of the integer pair (i, j), with $0 \leq i, j < N$ and $i \neq j$, so that each ordered pair occurs once. For $N = 1$, the stream ends immediately after h_1.

Each arc record starts with one ASCII line containing i, a comma, j, a comma and the number ϕ_{ij} of materialized knots. For example, `0,1,6` followed by LF introduces 6 knots on the arc from the depot to vertex 1. After this line, write ϕ_{ij} pairs of binary64 operands (x_k, y_k), alternating abscissa and ordinate for each knot in stored order. The first abscissa is h_0 and the last is h_1, while each ordinate is the absolute arrival time. Include every materialized knot, including collinear knots, identical consecutive pairs and repeated abscissas at either horizon

endpoint, since removing such knots or simplifying the sequence changes the encoded operands. At a repeated abscissa, the first ordinate is attained and the last supplies the right-hand limit when a following piece exists (Section 5.3). Reject missing, duplicate or self arcs and invalid knot sequences before accepting a digest.

Conformance vectors. For each input, the companion file `materialization-vectors.json` supplies the complete byte stream in hexadecimal and its expected digest. The 3-vertex matrix from `example-instance.json` occupies 99 bytes, and the TD vector `directed-jumps` occupies 180 bytes and covers both directions over $[0, 2]$. Its outward arc has repeated abscissas at 0, 1 and 2, while its return arc begins with the smallest positive binary64 subnormal, 2^{-1074}, as its arrival ordinate. Finally, the `depot-only` vector has a 40-byte stream consisting solely of the header and the horizon $[0, 4]$.

C Implementing the two exact binary64 profiles

A reference checker can turn every materialized binary64 operand f into a Python `Fraction.from_float(f)` and perform each operation on reduced fractions [Pytb]. Such a reference also provides a direct test of a faster backend. Given the same operands and routes, we compare feasibility, the objective's reduced numerator and positive denominator, and, for TD routes, both the minimum duration and the earliest departure attaining it. A displayed decimal cannot distinguish all these outcomes. The default backends are native, a C++ checker on a common dyadic scale for `binary64-exact-v1` and a C++ composition of route prefixes on exact rationals for `td-binary64-exact-v1`, while the Python `Fraction` reference remains selectable under both profiles. The two profiles permit different shortcuts because their route computations use different arithmetic operations. Table 3 lists the symbols of this appendix with the others, including those local to it.

C.1 Static routes on a common integer scale

Decode each operand once. The static checker sums arc costs, propagates times by addition and maximum, and tests inclusive bounds, so multiplying every arc cost, service duration and time-window bound by the same positive power of two preserves all these operations and comparisons. We can choose that scale exactly from the bits of the input. For a nonzero, nonnegative binary64 value f, let E and T denote its stored exponent and fraction fields, from which we recover an integer significand μ and a binary exponent η such that $f = \mu 2^{\eta}$ as follows.

$$(\mu, \eta) = \begin{cases} (2^{52} + T,\ E - 1023 - 52), & 1 \le E \le 2046, \\ (T,\ -1074), & E = 0. \end{cases}$$

The second line covers subnormals. Both signed zeros give zero units irrespective of the chosen scale, and the checker rejects a nonfinite or strictly negative cost or temporal operand. For positive μ, we strip its ν trailing zero bits and set $\mu' = \mu/2^{\nu}$ and $\eta' = \eta + \nu$. Among all nonzero operands, we then choose the binary-scale exponent b as the largest of 0 and the values $-\eta'$ (use $b = 0$ if all operands vanish), so that the integer representing f in units of 2^{-b}, called ticks, is exactly $\mu' 2^{\eta' + b}$. Since $\eta' + b \ge 0$, an integer left shift suffices even when a floating-point scale 2^b would overflow. The C++ checker implements this decomposition directly.

Evaluate a route in integer units. Consider 2 customers, a depot opening at 0, and successive arc costs 3/8, 5/4 and 1/2. The first customer opens at 1/2 and requires 1/4 unit of service, whereas the second closes at 2 and requires zero service. Take all remaining temporal bounds to be integral and nonbinding and all remaining arc costs to be integral, so that $b = 3$ and the common scale is 2^3. The vehicle reaches the first customer at tick 3, waits until tick 4

and leaves at tick 6. It then reaches the second customer at tick $6 + 10 = 16$, exactly its closing time, and returns to the depot at tick 20. Since the static objective counts travel alone, the checker returns $(3 + 10 + 4)/8 = 17/8$. In general, the sum in ticks is the integer $2^b C$ for the exact objective C, and the checker divides it by 2^b and reduces the resulting fraction before writing the result, while structural integers such as capacity and demand keep their original units.

Choose the accumulator width before evaluating submissions. A finite binary64 input can require more than 128 bits on the common scale. For example, a matrix containing both 2^{-1074} and 1 needs $b = 1074$, so its entry for 1 requires 1,075 bits. Moreover, even if every entry fits, an objective or arrival time can exceed the largest single operand. To bound these sums, let n be the number of customers, and let $c_{\max}$, $s_{\max}$, $e_{\max}$ and $l_{\max}$ be the largest arc cost, service duration, opening time and closing time, with zeros for absent temporal fields. The native checker computes, in arbitrary-precision integers, the bound in ticks

$$B = 2^b \max\{2nc_{\max},\ e_{\max} + (n+1)(c_{\max} + s_{\max}),\ l_{\max}\}.$$

To see why this bound suffices, note that a solution with $|\mathcal{S}|$ nonempty routes has $n + |\mathcal{S}| \le 2n$ arcs. A route has at most $n + 1$ arcs, and waiting can move its clock forward to an opening time of at most $e_{\max}$. Counting from its last wait, or from the depot opening if there was none, at most $n + 1$ arc and service increments remain. The time bound therefore covers every intermediate sum in a structurally valid route, including one that will miss a deadline.

For $B \le 2^{127} - 1$, the native checker uses unsigned 128-bit storage with the high bit reserved, checks each shift before executing it and, before each addition, checks that the second summand does not exceed $2^{127} - 1$ minus the first. A wider bound selects Boost arbitrary-precision integers at preparation, and a checked addition that exceeds the selected range triggers a retry in arbitrary precision. A backend without that fallback must signal an arithmetic range error so that the caller can choose another implementation. Treating that error as route infeasibility could discard a valid incumbent, for example a feasible route on the matrix above, whose sum in ticks exceeds $2^{127} - 1$ only because the common scale must also represent the entry 2^{-1074}. An exact total greater than the largest finite binary64 value is a different case, since it remains a valid objective whose binary64 display overflows (Section 6).

Pay for preparation where it is reused. With a prepared integer matrix, each route leg requires an indexed load and an integer addition, whereas with conversion on access the checker shifts the binary64 value when the route visits it. Both paths use the same b and return the same reduced result. Preparation therefore pays off when many submissions are checked on the same matrix, while conversion on access converts only the entries that the routes visit. A Python implementation can likewise prepare integer ticks using `float.as_integer_ratio()`

and Python's arbitrary-precision integers. When all operands are integral, $b = 0$ and these ticks are ordinary units, although the `binary64-exact-v1` identifier still describes their interpretation.

C.2 TD routes with arbitrary rationals

TD interpolation introduces divisions. If the endpoints of an ATF have binary64 coordinates $(0, 1)$ and $(6, 11)$, its slope is $5/3$, so even an integer departure at time 1 gives arrival $8/3$. Moreover, to find which initial departure reaches a window or an arc knot at a particular time, a checker must invert an affine function. Since a common power-of-two scale cannot represent all these intermediates, both C++ implementations convert each finite binary64 operand exactly to a rational using the GNU Multiple Precision Arithmetic Library (GMP). Their output consists of reduced integer pairs for duration and departure. The default backend composes the route prefix, and the event sweep described next is a second native traversal.

Compose the route prefix. After each visit, retain the vehicle clock as a function of the initial departure t, i.e. the ready-time function $\delta_{\mathbf{r}_j}$ of the prefix $\mathbf{r}_j = \langle v_1, \dots, v_j \rangle$, so that the final prefix gives $\delta_{\mathbf{r}_m} = \delta_{\mathbf{r}}$. Each retained knot holds its departure, the attained clock reading and the right-hand limit. If $x < x'$ are consecutive knot departures, $y^+(x) = \delta_{\mathbf{r}_j}(x^+)$ and $y^-(x') = \delta_{\mathbf{r}_j}(x')$, then a boundary at $y^+(x) < \zeta < y^-(x')$ is reached at

$$t_\zeta = x + \big(\zeta - y^+(x)\big)\frac{x' - x}{y^-(x') - y^+(x)}.$$

Inserting t_ζ when ζ is a knot of the next arc splits the prefix exactly where its formula changes. FIFO puts the encountered knots in increasing clock order, so the algorithm scans forward through the next arc as it processes the prefix. At a jump it evaluates the attained ordinate at the knot. A prefix that is constant at the jump then keeps using that same ordinate, whereas an increasing prefix that moves immediately past the jump uses the arc's right-hand limit. At a customer, trim the departure domain at the closing bound, insert a cut if the incoming clock crosses the opening time, then apply waiting and service. Before every arc, restrict the preceding clock to its departure horizon. Finally, a depot return deadline trims the final prefix once more.

Sweep through initial departures. The second algorithm visits the same kinds of boundaries in initial-departure order. Evaluate the route at the left endpoint and then, from a feasible t, trace a small positive displacement ε through the route. On the current open interval each clock has the form $y^+ + \gamma\varepsilon$, where y^+ is its right limit and $\gamma \geq 0$ its rate with respect to initial departure, so a positive rate reaches the next time boundary ζ after $(\zeta - y^+)/\gamma$. At a travel step, multiply the rate by the next ATF slope, and set it to zero when the vehicle waits until a fixed opening time. Record the smallest positive step allowed by an arc knot, a waiting threshold, a horizon or a deadline. The final clock remains affine until this first event, and its value at

the event follows by left continuity. Note, however, that a jump can make every departure immediately to its right infeasible even when the event itself is feasible.

Both algorithms consider feasible endpoints, including a singleton domain. On an open affine piece the duration is monotone or constant, and the lower value at an upward jump is attained, so no right limit can improve the best attainable value at that departure. Processing cuts in increasing order and replacing the best pair only for a strictly smaller duration retains the first departure on a minimizing plateau. As an independent check, the Python `Fraction` reference partitions the domain by boundary preimages, evaluates the forward route at the cuts, and checks the one-sided limits. A Python event-sweep variant uses the same rational type with the second traversal.

A jump reached after waiting. Take one customer with service window $[2, 6]$, zero service and no depot window, and let both arcs have departure horizon $[0, 6]$. We use the outbound ATF above and give the return arc knots $(0, 1)$, $(2, 3)$, $(2, 4)$ and $(6, 8)$. Since the outbound arrival $1 + 5t/3$ first reaches 2 at $t = 3/5$ and reaches 6 at $t = 3$, the feasible initial departures are $[0, 3]$. For $0 \le t \le 3/5$, waiting fixes the customer's departure at 2, and the attained return-arc ordinate is 3 throughout this plateau, including its right endpoint. For $3/5 < t \le 3$, the return arc instead adds 2 time units. Thus for every feasible departure the duration is

$$\delta_{\mathbf{r}}(t) - t = \begin{cases} 3 - t, & 0 \le t \le 3/5, \\ 3 + 2t/3, & 3/5 < t \le 3. \end{cases}$$

Hence the reported duration is $12/5$ and the earliest departure is $3/5$. Prefix composition inserts the exact preimage $3/5$ of the return-arc knot, while the event sweep reaches it from 0 when waiting ends. By contrast, a traversal that uses the upper ordinate 4 while the vehicle waits at 2 changes the optimum.

Scope of the native implementations. The native static checker checks complete solutions. Demands and capacities must be nonnegative integers representable both in signed 64-bit storage and exactly in binary64, and its input adapter enforces zero depot service. The native TD routines, by contrast, check one prepared route at a time, while the calling solution evaluator checks coverage, fleet and capacity, sums the exact durations and adds any vehicle charge. Either native TD routine can therefore reuse the aggregation and serialization of the Python reference. These native backends exist because a curator or a search repeats the same exact checks on many routes, and fixed-width integers or GMP rationals perform them more cheaply than generic Python fractions while returning the same reduced results.

D Alternative input readings and arithmetic modes

Three choices can be varied independently. A reader can assign another numerical meaning to a published token, an evaluator can round an intermediate result, or an implementation can compute the same exact operation in another representation. Of these, only the first two can change a feasible schedule or an objective. The following identifiers let a reader interpret a result with an explicitly declared alternative profile, whereas the default static and TD rules in the body apply to the binary64 operands.

D.1 Integral and decimal readings

The static profile `integer-exact-v1` requires integral binary64 arc and temporal operands throughout the instance, and for eligible DIMACS and ORTEC instances exact integer evaluation gives the same costs and feasibility decisions as `binary64-exact-v1`. By contrast, a common dyadic scale inside a fast binary64 checker is an implementation choice, which also accommodates fractional binary64 values.

The `decimal-exact-v1` profile reads the original decimal token into an exact rational before any binary64 conversion, and the `decimal-grid-3-v1` profile additionally requires every such arc and temporal operand to be a multiple of 10^{-3}. For example, multiplying the exact decimal value of `215.359` by 1000 gives 215359. A matrix generator may use a 3-decimal quantization and still publish a benchmark whose default profile reads the emitted token as binary64, so the declared reading determines which value the checker must compare to an exact deadline.

For $x \in \mathbb{Q}$ and decimal precision $d \in \mathbb{N}_0$, define

$$Q_d(x) = 10^{-d}\,\mathrm{roundEven}(10^d x) \in \mathbb{G}_d, \qquad \mathbb{G}_d = 10^{-d}\mathbb{Z}. \tag{15}$$

At an exact half-integer, roundEven selects the even integer. Floor quantization is $10^{-d}\lfloor 10^d x \rfloor$, whereas toward-zero quantization discards the fractional part of $10^d x$. Floor and toward zero agree for the nonnegative arc quantities here, but are distinct operations in general. Generation code and its documentation MUST identify the rule and the stage at which it acts. Loading an already published grid token MUST NOT quantize it again.

Algorithm 2 Exact decimal-grid quantization

Require: $x \in \mathbb{Q}$, decimal precision $d \in \mathbb{N}_0$
1: Choose $(p, q) \in \mathbb{Z} \times \mathbb{Z}_{>0}$ with $p/q = 10^d x$
2: $(\ell, \rho) \leftarrow \mathrm{divmod}(p, q)$, where $p = \ell q + \rho$ and $0 \le \rho < q$
3: **if** $2\rho > q$ or ($2\rho = q$ and ℓ is odd) **then**
4: $\quad \ell \leftarrow \ell + 1$
5: **end if**
6: **return** $\ell / 10^d \in \mathbb{G}_d$

The reference implements this definition as `quantize_decimal`. Python's `round(x, 3)`, by contrast, acts on the actual binary64 value of x, so its decimal result can differ from rounding a desired exact decimal input. We therefore record the retained historical recipe as the actual call, rather than silently replacing it with decimal-first quantization [Pytd].

D.2 Rounded and historical evaluation

The static companion also supports three binary64 arithmetic variants, each of which reads the same binary64 inputs, rounds after every addition and keeps the prescribed order. They differ in the order and grouping that they prescribe. `canonical-fp64-flat-v1` adds each arc to one accumulator across the canonically ordered routes, `canonical-fp64-grouped-v1` rounds within each route and then adds the rounded subtotals, and `submitted-fp64-flat-v1` visits the routes in their submitted order. A stored calculation using such a profile retains that identity. For TD results from released checkers, `td-fold/1` and `td-fold/2` identify the historical evaluations. A solver claim attached to a result targets the profile under which its routes were evaluated, as Appendix G explains.

E Attainment of the minimum route duration

Fix a customer order and its ATFs under Section 5.3, and let $D_{\mathbf{r}}$ denote the set of feasible depot departures. Its initial domain is the intersection of a closed departure horizon and any depot departure window. On this interval the initial time map is the identity. A FIFO arc with the lower value attained at each jump is nondecreasing and left-continuous, and the customer map $u \mapsto \max(u, e_j) + s_j$ has the same two properties. Composing two such maps preserves both properties on the feasible domain, since the inner map approaches its value from below or is constant.

Nonnegative travel and service preserve the lower horizon bound at each step, whereas a latest-service constraint, the upper bound on an arc departure or a depot return deadline restricts a prefix time to a closed upper bound. By monotonicity, the preimage of such a bound within the current domain is an initial interval, and left continuity includes its right endpoint whenever that interval is nonempty. By induction, $D_{\mathbf{r}}$ is therefore empty or a compact interval, possibly reduced to a singleton.

A nondecreasing left-continuous return map can jump only upward immediately to the right of a point, which makes it lower-semicontinuous. Subtracting the continuous departure time preserves this property, so the duration attains its minimum on every nonempty $D_{\mathbf{r}}$, and its set of minimizers is compact and has a smallest member.

All knots, service durations and window bounds are rational. Within each finite affine prefix piece, the preimage of a boundary is either a rational cut, the entire piece or the empty set. There are consequently finitely many rational cuts, with an affine duration on each intervening open interval. A nonconstant affine piece has no interior minimum, while on a constant minimizing piece lower semicontinuity and closedness of the feasible domain include its left endpoint among the minimizers. The earliest minimizer is therefore a rational cut or domain endpoint, and evaluating the route there yields a rational duration.

F A departure at a jump of the return arc

The jump convention of Section 5.3 decides the duration of a route whose schedule reaches a repeated abscissa exactly. Consider the customer sequence $\langle 23, 21, 12, 3, 4, 25, 30, 29\rangle$ on the TDVRP instance Rifki-30 of the Rifki2020 family, whose times are in seconds. A vehicle leaving the depot at 1341 is ready to leave customer 29 at exactly 2880, where the return arc $(29, 0)$ stores 2 knots with the same abscissa. The first knot gives the arrival 3084, which is attained at that instant, while the second gives the upper ordinate 3126, from which the following piece starts (Figure 1).

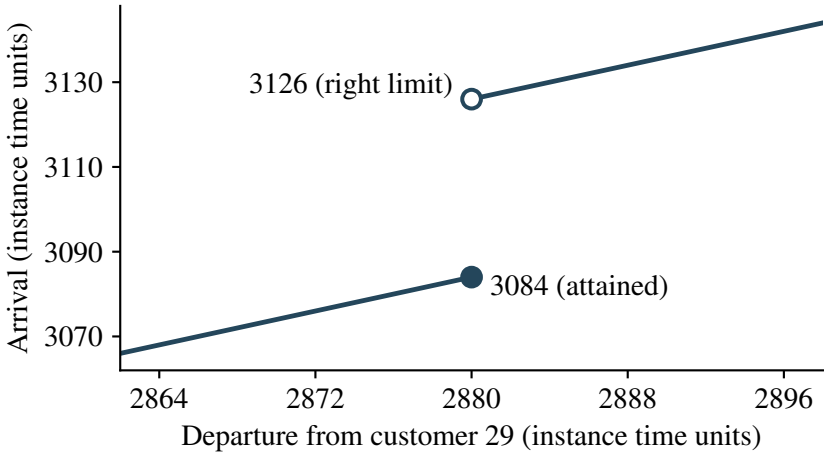

Figure 1: Final arc $(29, 0)$ of the Rifki-30 route near departure 2880. The filled point is the arrival attained at the repeated abscissa, and the open point marks the upper ordinate from which the next piece starts.

Under `td-binary64-exact-v1`, the vehicle therefore returns at 3084, and the route has duration $3084 - 1341 = 1743$. This value is also the minimum duration over all feasible depot departures, and 1341 is the earliest departure attaining it. A vehicle ready to leave customer 29 at any later time, however close to 2880, would instead follow the upper piece and return after 3126. Hence, an evaluation that rounds the ready time upwards by a single binary64 step crosses the jump and adds slightly more than 42 to the return time, which is why the exact profile keeps every intermediate time exact.

G Solver self-reports

A stored BKS records the checker's verdict on its routes in `metadata.standard.evaluation`, as described in Section 6. A solver can also state that the incumbent it returns is optimal, for instance when an exact search closes its gap, and the submitter may then record that statement in the optional object `metadata.optimality`. Its status `solver-reported-optimal` says what kind of statement it is, namely a self-report attributed to the solver that made it. The claim is scoped to the model and profile that the solver names, and the checker neither repeats the search nor verifies the bound behind it. Verification of the routes and the claim of optimality therefore remain two distinct statements, made by two different programs. A record without a claim simply omits the object.

The object has three required fields. Its `status` is always `solver-reported-optimal`, its `target_profile` names the evaluation profile under which the solver claims optimality, and its `solver` field identifies the solver together with its version or revision when known. An optional `campaign` field may point to the run or to its preserved evidence, and further fields may record the optimum and the dual bound reported by the solver, provided that their values and meaning can be established. Since an optimality claim compares the incumbent with every other feasible solution of the same numerical problem, its target MUST be the profile under which the stored routes were evaluated. The reference library accordingly refuses to record a new claim whose `target_profile` differs from the profile of the evaluated family, for static and TD results alike. Appendix D lists the other readings and arithmetic modes that a checker may support.

For example, suppose that a fictitious solver `ToyBPC 0.1` solves the companion TD instance of Section 6 under `FleetCostDuration` and reports that the two routes `[1]` and `[2]`, of exact cost 79/6, form an optimal solution. The checker verifies these routes as for any submission, and the stored result then carries the following object next to `metadata.standard`. The distributed file `example-td-result.json` carries no claim, so this excerpt only illustrates the format.

```
"optimality": {
  "status": "solver-reported-optimal",
  "target_profile": "td-binary64-exact-v1",
  "solver": "ToyBPC 0.1"
}
```

From such a record, a reader may conclude that the routes are feasible and cost exactly 79/6 under the declared benchmark edition, because the checker established this, and that the named solver reported finding no better solution under `td-binary64-exact-v1`. The optimality itself rests on the correctness of that solver and of its own arithmetic, which the checker does not audit. A reader who needs a proof therefore treats the claim as a pointer to the evidence of the run. Note that a later submission strictly improving the exact key under the same edition,

objective and profile would show the claim to be wrong.

The objective reported with an incumbent

A solver can report the cost it assigned to a feasible solution under its own search model, whether or not it claims optimality. We record such a report in `metadata.solver_report`, where `objective` names the reported quantity, `value` preserves the report and `encoding` tells the reader how to interpret it. An `integer` or `decimal-text` report keeps the original token as a string, including any printed exponent or trailing zeros, whereas a `binary64` report from a solver API uses a finite JSON number with a round-trip spelling. For a known scale, `family_units_per_solver_unit` stores the positive rational number of family cost units represented by one solver unit, using the fraction encoding of Section 6.

When the report gives a routing cost and its unit is known, `metadata.solver_cost` displays that cost in family units. To obtain it, multiply the reported value by the unit exactly and round once to nearest binary64, ties to even. Omit `solver_cost` if the routing cost or the conversion is unknown, or if its projection overflows. This display lets a reader set the solver's own view of an incumbent beside the checked value, while the comparison of incumbents always uses the exact cost in `metadata.standard`.

For example, an LKH-3 run on a Toronto instance reports the integer `"65617528"` for `total-arc-cost`. Since its unit is 1/100, `metadata.solver_cost` displays $\mathrm{RN}_{64}(65617528/100)$ as `656175.28`. An exact evaluation of the same routes on the family matrix, however, gives 46174228443702207525/70368744177664, whose binary64 display is `656175.252`, and the BKS comparison uses this latter rational. The two values differ because LKH-3 searched on integer-scaled arc costs in units of 1/100, whereas the checker evaluates the binary64 values of the family matrix exactly.

A solver may also optimize a composite objective. For example, PyVRP can report `vehicle-penalized-cost` and separately report the `total-arc-cost` of the routes, in which case `routing_cost_component` records the latter quantity with its own `objective`, `value` and `encoding`, in the unit of the report, and `solver_cost` is computed from this component. Kayros can instead report `total-route-duration` directly. An ordinary solver report can accompany any incumbent, whereas a solver that claims optimality identifies the target of that claim separately in `metadata.optimality`.

H Certifying an instance edition and a route result

Consider a family that distributes travel times for a set of customer locations, and a team that submits a shorter tour a month after the instance is published. The new tour changes the incumbent, while the locations, travel times and rules under which the two tours are compared stay fixed. An instance edition therefore has its own identity, which the family can publish before it receives a solution. We describe below how to publish the inputs, then the manifest, and finally how to bind a result to them.

Publish the inputs first. The publisher assigns an edition label such as `r1` to an instance variant within a family, and the descriptor and required sidecars are available as files with content hashes. For regenerated operands, the publisher also preserves the source distribution containing the materialization function, its parameters and the relevant environment information. These artifacts must remain retrievable by their recorded digests. Once these artifacts are in place, the complete matrix or ATF stream is computed and hashed as specified in Appendix B. If the family supplies both a stored representation and a regeneration procedure for the same arc data, a verifier can compare the two stream digests directly.

Publish the manifest second. For a descriptor named `C101.vrp.json`, the edition manifest is `C101.manifest.json`, located in the same directory. The manifest records the edition label in `benchmark_revision`, the objective and numerical profile, the input references, and the expected materialization digest. Once these fields are fixed, the publisher preserves the exact manifest bytes. Changing an input, the objective or the prescribed recipe therefore requires a new edition of that variant, while previously issued manifests and their inputs must remain available for replay.

Bind the result last. The verifier reads the manifest, checks the input and materialization digests, and evaluates the submitted routes under the declared objective and profile. The result then records the edition label and the SHA-256 of the complete manifest file in `metadata.standard`. This digest includes the manifest's whitespace and final newline, and it appears in the result, outside the bytes being hashed. Because the manifest refers only to inputs and materialization artifacts, replacing a BKS or verifying it with another conforming backend leaves the manifest intact. A solver claim attached to the result names its own target profile, as described in Appendix G.

Exact evaluation before certification. A family may receive routes before it publishes the manifest of an edition. A checker can still evaluate them exactly under the family profile and record the outcome, which the MAMUT-routing library stores apart from the certified record,

under `metadata.exact_evaluation`. Such a record carries the same exact objective and route witnesses, but it is not bound to an edition. The certified record `metadata.standard` exists only once the manifest of the immutable edition has been verified, because the identity fields that bind the result to its inputs come from that manifest. Note that the library also keeps an evaluation under an objective other than the one declared by the manifest in this unbound form.

Verification after a repository move. A Git commit identifies a repository snapshot together with its history, so replaying the same files on a different parent can change the commit identifier while preserving each file byte for byte. A release archive, for its part, may contain the same files without any Git metadata. In both cases, the verifier can follow the result's manifest digest, check the descriptor and sidecars by their file digests, and reproduce the matrix or ATF digest. While a version-control reference can help a reader locate a source, certification depends on the preserved artifact bytes. For example, moving a copied instance and its manifest into a directory outside a checkout preserves their identity. By contrast, editing a single customer deadline in that copy invalidates the descriptor hash, even if every entry in the travel matrix remains identical.

Licence and AI-use statement

This document is distributed under the Creative Commons Attribution 4.0 International licence (CC BY 4.0). The companion code and example files of its repository are distributed under the MIT License.

AI assistants were used for source inspection, for drafting and checking the text and for preparing the companion code, under the supervision of the authors. The authors reviewed the content and are responsible for it.