Customizable Route Planning in the Browser

A reference implementation and measurement of partition-and-overlay shortest paths on real road-network data

Ryan Tenney
ryan.tenney.co

August 2026

Abstract

Customizable Route Planning (CRP) is the shortest-path technique that answers point-to-point queries on a road network whose edge weights change every few minutes. It splits preprocessing in two: an expensive, rarely repeated phase that depends only on the road *topology*, and a cheap, frequently repeated phase that absorbs a new *metric*. This paper documents a complete, self-contained implementation of all three CRP phases — partition, customization, and query — written in TypeScript and executed entirely in a web browser on an OpenStreetMap extract of downtown Chicago (4,654 junction vertices, 9,231 directed edges, 501.7 km of road). We state and prove the exactness of the overlay decomposition, give an admissible and consistent heuristic that makes the overlay search goal-directed without sacrificing exactness, and describe the path-unpacking step that recovers road geometry from a chain of summarized boundary hops. The implementation computes entirely in integers — whole seconds and centimeters — and we show that the resulting floored geometric potential is still exactly consistent, so nothing is given up for the reproducibility that integer arithmetic buys. We then measure the implementation: across 2,700 random queries spanning nine partition granularities, the CRP query is exact on every single query (cost deviation from a full-graph A* baseline: exactly zero, as an integer comparison rather than a tolerance), and it settles 2.8× fewer vertices at the demo's operating point. We also report the result that matters for anyone considering CRP at small scale: on a graph this size the settle-count reduction does *not* translate into a comparable wall-clock speedup, because each overlay settle expands a dense clique row. We characterize the cost model that explains this, locate the empirical optimum (≈ 291 vertices per cell on this graph, where CRP is 1.17× faster), and identify the regime in which the asymptotic argument for CRP takes over.

Contents

1 Introduction

1.1 The production problem

Between 2018 and 2026 I was the technical lead for Pathfinder, the vehicle routing platform at Grubhub. Pathfinder estimates drive times for every stage of a delivery: which restaurants a diner is shown, which courier is offered which order, when a kitchen should start cooking, and when the food is promised to arrive. That workload is roughly 1.5 billion shortest-path problems per day. Two properties of the workload jointly rule out most of the textbook answers:

1. **Latency.** Each query must return in single-digit milliseconds, because hundreds of them sit behind one user-visible action (ranking a page of merchants, scoring a batch of courier–order assignments).

2. **Volatility.** The edge weights are travel times under live traffic. They are re-estimated every few minutes. A technique whose preprocessing bakes in the weights must redo that preprocessing on the same cadence, or serve stale answers.

Dijkstra's algorithm [7] satisfies volatility perfectly — it uses the weights as it finds them — and fails latency badly. It is *label-setting*: to answer a query it settles every vertex closer to the source than the target is. In a metro area, a 30-minute query settles most of the graph.

Contraction Hierarchies [3] invert the trade. Preprocessing inserts shortcut edges in a weight-dependent vertex order, after which queries take microseconds — but the shortcuts and the order are functions of the metric. Rebuilding them on every traffic update is far too slow. Hub labeling [4] is faster still at query time and even more metric-committed.

Customizable Route Planning (CRP), developed at Microsoft Research for Bing Maps [1], is the technique designed for exactly this shape. It observes that a road network has two kinds of structure: its *topology*, which changes when road crews change it, and its *metric*, which changes with the weather. CRP spends the expensive preprocessing on the topology and makes the metric-dependent step cheap enough to re-run continuously.

1.2 What this paper is

This paper documents a from-scratch implementation of CRP that runs in a web browser — graph construction, partitioning, customization, query, and path unpacking, in TypeScript, on real OpenStreetMap data, with no server component. It accompanies an interactive demo[1] in which each phase is animated as it executes.

The contributions are:

- A precise statement and proof of the overlay decomposition (Section 6), including the admissibility and consistency of the goal-directed heuristic used on the overlay graph (Lemma 5), which is what lets the implementation be both goal-directed and exact.
- An account of the engineering that makes the technique viable inside a single browser thread (Section 7): compressed-sparse-row adjacency, epoch-stamped scratch buffers, and flat-array binary heaps, with no allocation on any hot path.
- An all-integer formulation (Section 2.2) — whole seconds and centimeters — together with a proof that the doubly-floored geometric potential remains consistent (Lemma 6), so route costs are exactly reproducible and the exactness check against the baseline is an integer equality rather than a floating-point tolerance.
- A measurement study (Section 8) over 2,700 queries and nine partition granularities, which confirms exactness on every query, locates the empirical optimum for cell size, and reports honestly that the wall-clock speedup at this graph scale is small even though the settle-count reduction is large — with a cost model that explains why.

Section 10 enumerates the deliberate simplifications relative to a production deployment, so that no number here is mistaken for a claim about a production system.

2 Preliminaries

2.1 Notation

A road network is a directed graph $G = (V, E)$ with $n = |V|$, $m = |E|$, and a weight function $w : E \to \mathbb{R}_{>0}$ giving traversal time in seconds. The pair (G, w) is the *metric*; G alone is the *topology*. $\mathrm{dist}_w(s, t)$ denotes the minimum weight over all $s \to t$ paths. Each vertex v carries a plane position $p(v) \in \mathbb{R}^2$ in meters (an equirectangular projection about the extract's center latitude; the distortion is well under 1% at city scale). We write $\|u - v\|$ for the Euclidean distance between $p(u)$ and $p(v)$, and

$$v_{\max} \;=\; \max_{e \in E} \frac{\mathrm{len}(e)}{w(e)}$$

for the fastest speed realized anywhere in the network, computed once from the segment geometry.

2.2 Units: everything is an integer

The implementation computes in integers throughout. Lengths and coordinates are whole *centimeters*; travel times are whole *seconds*; unreachability is a sentinel value $(2^{31} - 1)$ rather than a floating-point infinity. Floating point survives only in three places that never feed a route cost: the projection from latitude/longitude, the one square root per distance evaluation (rounded away immediately), and the presentation layer — canvas transforms, colors, animation clocks.

This is not fastidiousness. Three things follow from it:

[1] `https://ryan.tenney.co/projects/crp`

- **Costs are associative.** A route's cost is the sum of its segments in any order. With floats, two decompositions of the same path — one summed edge by edge, one assembled from clique entries — can differ in the last bits, and the difference is not a bug you can fix, only one you can bound.
- **The exactness check is an equality.** Section 8 compares every CRP query against a full-graph baseline. In integers that comparison is $=$; in floats it would be $|\cdot| < \varepsilon$ for some ε chosen by the person who wants the test to pass.
- **Results are reproducible.** Heap order is determined by integer keys, so two runs of the same query settle the same vertices in the same sequence, on any machine.

The rounding happens once, at graph construction: a segment's length is its polyline length in centimeters, and its weight is $\max(1, \mathrm{round}(\mathrm{len}/v))$ seconds. The floor of one second matters for 113 of the extract's 6,275 segments (1.8%), all of them very short; it exists because a zero-weight edge would let a route cross a junction for free. Section 8.4 measures what the quantization costs.

2.3 Dijkstra and A*

Dijkstra's algorithm settles vertices in nondecreasing order of $\mathrm{dist}(s, \cdot)$ and runs in $O(m + n \log n)$ with a Fibonacci heap, $O(m \log n)$ with the binary heap used here. A* [5] adds a potential function h and orders the queue by $f(v) = d(v) + h(v)$. Two standard properties matter below.

Definition 1. h is *admissible* for target t if $h(v) \le \mathrm{dist}(v, t)$ for all v, and *consistent* if $h(u) \le w(u, v) + h(v)$ for every edge (u, v).

Consistency implies admissibility (given $h(t) = 0$) and additionally guarantees that no vertex is settled twice, so A* with a consistent h is exactly Dijkstra on the reduced weights $w'(u, v) = w(u, v) - h(u) + h(v) \ge 0$. Throughout we use the geometric lower bound

$$h(v) \;=\; \left\lfloor \frac{\lfloor \|p(v) - p(t)\| \rfloor}{v_{\max}} \right\rfloor, \tag{1}$$

the time to fly straight to the target at the fastest speed the network permits — which can never overestimate an actual driving time — floored to whole seconds so that it, like every other quantity, is an integer. Here $v_{\max}$ is rounded *up* to a whole number of centimeters per second, since dividing by a larger speed can only shrink the estimate. Lemma 6 shows the two floors cost nothing: the integer potential is exactly as consistent as the real-valued one.

3 Building the routing graph

The input is an OpenStreetMap extract of a 5.8×6.7 km window of downtown Chicago (bounding box 41.85–41.91°N, −87.67–−87.60°E), fetched once via the Overpass API and committed to the repository so that builds are reproducible and offline. It contains 20,060 nodes and 5,261 highway ways.

OSM ways are not a routing graph. Four transformations produce one.

Junction detection. A raw node becomes a graph vertex when it is a way endpoint or is shared by two or more way traversals. Interior nodes exist only to describe curvature.

Chain contraction. Each way is cut at its junction nodes into *segments*. The intervening degree-two nodes are folded into the segment's polyline geometry, which is kept for rendering and for the unpacking step of Section 6.4 but is invisible to the search. This is the single largest reduction: 20,060 raw nodes collapse to 4,654 junction vertices.

Quantity	Value
Raw OSM nodes	20,060
Raw OSM ways	5,261
Junction vertices n	4,654
Undirected segments	6,275
Directed edges m	9,231
Total road length	501.7 km
Largest SCC (of junction vertices)	96.6%
Segments at the 1-second weight floor	113 (1.8%)
Graph build time	30 ms

Table 1: The routing graph, built from the committed OSM extract.

Travel times. Segment weight is $\max(1, \text{round}(\text{len}/v))$ seconds, where v comes from the OSM `maxspeed` tag when present and otherwise from a table of per-highway-class free-flow defaults (96 km/h motorway, 48 km/h primary, 28 km/h residential, and so on), converted to whole centimeters per second. All speeds are multiplied by a fixed congestion factor of 0.45, which stands in for the live traffic metric a production customization would supply. This is where the model stops being real-valued: latitudes, speed tags, and the projection are continuous, and everything downstream of this line is an integer.

Strong connectivity. One-way tags and the extract's clipped border strand pockets of vertices that cannot be left or cannot be reached. Because a query between such a pocket and the rest of the map has no answer at all, the implementation extracts the largest strongly connected component with an iterative Kosaraju–Sharir double pass [8] and discards the rest; 96.6% of junction vertices survive.

The result is stored as two compressed-sparse-row (CSR) adjacency structures, forward and reverse, since the query needs a backward search from the target. Table 1 gives the resulting sizes.

4 Phase I: Partition (metric-independent)

Definition 2. A *partition* of G is a map $\text{cell} : V \to \{1, \ldots, k\}$. An edge (u, v) with $\text{cell}(u) \neq \text{cell}(v)$ is a *cut edge*; its endpoints are *boundary vertices*. B_c denotes the boundary vertices of cell c, and $B = \bigcup_c B_c$.

The partition is the only structure CRP derives from topology alone, and everything downstream inherits its quality. Two objectives are in tension: cells should be *balanced* (so that no single cell dominates customization or query cost) and the *cut* should be small (because, as Section 5 shows, cost is quadratic in the number of boundary vertices per cell). Balanced graph partitioning is NP-hard, so every implementation is a heuristic. Production CRP uses PUNCH [2], which finds "natural cuts" — rivers, rail lines, highways — that road networks conveniently possess.

This implementation uses recursive geometric bisection, chosen because it is simple, fast, deterministic, and animates legibly. Repeatedly: take the largest cell; project its vertices onto each of four candidate axes (0°, 90°, 45°, 135°); split at the median along each, which enforces perfect balance by construction; count how many intra-cell edges each candidate severs; and keep the cheapest. Splitting the largest cell $k - 1$ times costs $O(k\, n \log n)$ overall in the worst case, and 25 ms in practice for $k = 48$.

The $\sqrt{k}$ growth visible in Table 2 is the reason CRP works at all: doubling the cell count

Cells k	Vertices/cell	Cut segments	$\lvert B\rvert$	$\lvert B_c\rvert$ avg
8	582	210	380	47.5
16	291	338	608	38.0
32	145	513	918	28.7
48	97	648	1,141	23.8
64	73	754	1,314	20.5
128	36	1,082	1,798	14.0
256	18	1,555	2,449	9.6

Table 2: Partition quality versus granularity. The total boundary $|B|$ grows roughly as $\sqrt{k}$ — the perimeter-to-area law of planar subdivisions — while the per-cell boundary $|B_c|$, which drives the quadratic customization cost, shrinks.

does not double the boundary. Cells are areas, cuts are perimeters, and perimeter grows as the square root of area.

5 Phase II: Customization (metric-dependent)

Definition 3. The *clique* of cell c is the matrix $M_c \in (\mathbb{R}_{>0} \cup \{\infty\})^{|B_c|\times|B_c|}$ with

$$M_c[i][j] \;=\; \mathrm{dist}_w^c(b_i, b_j)\,,$$

the cost of the shortest $b_i \to b_j$ path that never leaves cell c (∞ if none exists).

The *overlay graph* H has vertex set B and two kinds of edge: a clique edge $b_i \to b_j$ of weight $M_c[i][j]$ for every ordered pair of boundary vertices in a common cell, and the original cut edges between cells. H is a faithful summary of G for the purpose of travelling between boundary vertices, and it is dramatically smaller: 1,141 vertices instead of 4,654, with the interior of every cell compressed into a table.

Customization computes M_c for all c: for each $b \in B_c$, run a Dijkstra from b restricted to cell c (relaxing an edge only when its head is also in c), then read off the distances to the other members of B_c. Nothing about this touches another cell, which is the property that makes customization *embarrassingly parallel* — in production, sharded across cores and machines, so a new traffic metric propagates in seconds.

Cost. With n_c vertices and m_c edges in cell c, customization costs

$$\sum_c |B_c| \cdot O(m_c \log n_c) \qquad \text{time, and} \qquad \sum_c |B_c|^2 \quad \text{entries of memory.}$$

Both terms pull toward small cells, while query cost (Section 6) pulls the other way; Section 8.5 measures where the two meet. At $k = 48$ the implementation runs 1,141 cell-confined Dijkstras producing 29,757 clique entries — 116 KiB at four bytes per entry — in 8 ms. This is the number that matters operationally: it is the cost of absorbing a new metric, and it is three orders of magnitude below the cost of rebuilding a contraction hierarchy.

Two matrices. Each clique is stored twice, row-major and transposed. A forward overlay search reads row i of M_c contiguously; a backward search reads column i, which is row i of M_c^{T}. The duplicate costs memory that is already small and buys cache-contiguous access in both directions.

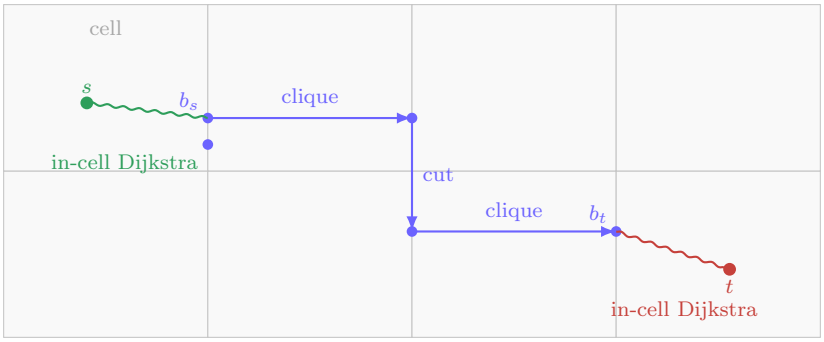

Figure 1: The three-phase query. The source-cell search prices every way out of cell(s); the overlay search hops between boundary vertices along clique entries (within a cell) and cut edges (between cells); the target-cell backward search prices every way in. Cell interiors are never traversed by the overlay search — they were summarized during customization.

6 Phase III: Query

6.1 The decomposition

A point-to-point query runs three searches instead of one:

1. a forward Dijkstra from s *confined to* cell(s), pricing every way out of the source cell;
2. an A* search over the overlay graph H, hopping between boundary vertices along clique entries (through a cell) and cut edges (between cells);
3. a backward Dijkstra from t confined to cell(t), pricing every way in.

Theorem 4 (Exactness)**.** *Let s and t lie in different cells. Then*

$$\mathrm{dist}_w(s,t) \;=\; \min_{b_s \in B_{\mathrm{cell}(s)},\, b_t \in B_{\mathrm{cell}(t)}} \Big[\, d_{\mathrm{cell}}(s \to b_s) \;+\; d_H(b_s \to b_t) \;+\; d_{\mathrm{cell}}(b_t \to t) \,\Big],$$

where d_{cell} denotes a distance realized without leaving the cell in question and d_H denotes distance in the overlay graph.

Proof. ($\geq$) Let P be a shortest $s \to t$ path. Since $\mathrm{cell}(s) \neq \mathrm{cell}(t)$, P crosses at least one cut edge. Let b_s be the tail of the first cut edge on P and b_t the head of the last; both are boundary vertices, of cell(s) and cell(t) respectively. Decompose P at these points. The prefix $s \to b_s$ lies wholly in cell(s) by choice of the first crossing, so its weight is at least $d_{\mathrm{cell}}(s \to b_s)$, and symmetrically for the suffix. The middle portion $b_s \to b_t$ alternates between maximal subpaths interior to a single cell (each running between two boundary vertices of that cell, hence of weight at least the corresponding clique entry) and single cut edges (present in H verbatim). Concatenating those bounds gives a $b_s \to b_t$ walk in H of weight at most $w(P|_{b_s \to b_t})$, so $d_H(b_s \to b_t) \leq w(P|_{b_s \to b_t})$. Summing the three bounds shows the right-hand side does not exceed $w(P) = \mathrm{dist}_w(s,t)$.

($\leq$) Conversely, every term on the right corresponds to a real walk in G: d_{cell} values are weights of actual in-cell paths, clique entries are weights of actual in-cell paths between boundary vertices, and cut edges are actual edges. Their concatenation is an $s \to t$ walk in G, whose weight is therefore at least $\mathrm{dist}_w(s,t)$. Since all weights are positive, a walk's weight is at least that of the shortest path. □

The same-cell case is handled separately: the source-cell search may already reach t directly, and that candidate is seeded as the incumbent before the overlay search begins. It is not sufficient on its own — the shortest $s \to t$ route may well leave the cell and come back — so the overlay search still runs, and the better of the two wins.

6.2 Goal direction on the overlay

Searching H with plain Dijkstra floods outward in all directions. The implementation instead runs A* with the geometric potential of Equation (1), evaluated at the boundary vertices' own positions.

Lemma 5. *$h(b) = \|p(b) - p(t)\|/v_{\max}$ is consistent on the overlay graph H.*

Proof. Let (u, v) be an edge of H with weight ω. If it is a cut edge, $\omega = \text{len}(u, v)/\text{speed} \geq \|p(u) - p(v)\|/v_{\max}$ by the definition of $v_{\max}$ and because a polyline is at least as long as its chord. If it is a clique entry, ω is the weight of an actual path in G from u to v, whose length is at least $\|p(u) - p(v)\|$ and each of whose edges is traversed at speed at most $v_{\max}$; hence again $\omega \geq \|p(u) - p(v)\|/v_{\max}$. By the triangle inequality, $\|p(u) - p(t)\| \leq \|p(u) - p(v)\| + \|p(v) - p(t)\|$, so $h(u) \leq \omega + h(v)$. □

Lemma 5 is stated for the real-valued potential. What the implementation actually evaluates is the doubly-floored integer of Equation (1), and flooring a potential is not generally harmless — it can break the very inequality consistency asserts. Here it does not.

Lemma 6. *Let $g(v) = \|p(v) - p(t)\|$ and let $v_{\max}$ be a positive integer. Then the implementation's potential satisfies $h(v) = \lfloor \lfloor g(v) \rfloor / v_{\max} \rfloor = \lfloor g(v)/v_{\max} \rfloor$, and h is consistent on H.*

Proof. The first equality is the nested-floor identity $\lfloor \lfloor x \rfloor / m \rfloor = \lfloor x/m \rfloor$, valid for real $x \geq 0$ and any positive *integer* m — which is why $v_{\max}$ is rounded to an integer rather than kept as a ratio. So the inner floor, which is what the code computes when it takes an integer square root, costs nothing at all.

For consistency, let (u, v) be an edge of H with integer weight ω. By Lemma 5's argument, $\omega \geq \|p(u) - p(v)\|/v_{\max}$, and by the triangle inequality $g(u) \leq \|p(u) - p(v)\| + g(v)$. Dividing by $v_{\max}$,

$$\frac{g(u)}{v_{\max}} \;\leq\; \omega + \frac{g(v)}{v_{\max}}.$$

Taking floors of both sides and using that ω is an integer,

$$h(u) \;=\; \left\lfloor \frac{g(u)}{v_{\max}} \right\rfloor \;\leq\; \left\lfloor \omega + \frac{g(v)}{v_{\max}} \right\rfloor \;=\; \omega + \left\lfloor \frac{g(v)}{v_{\max}} \right\rfloor \;=\; \omega + h(v). \qquad \square$$

The integer square root deserves one remark, since the argument above assumes $\lfloor g \rfloor$ is computed exactly. The implementation evaluates $\lfloor \sqrt{\Delta x^2 + \Delta y^2} \rfloor$ in double precision. The radicand is an integer below 2^{53}, hence exact; IEEE 754 square root is correctly rounded; and the true root is irrational unless the radicand is a perfect square, in which case the root is exact. So the computed value never lands on the wrong side of an integer boundary, and the floor is the true one.

Consistency has three consequences: the search never settles a boundary vertex twice, exactness is preserved (with Theorem 4), and goal direction is therefore free — the frontier leans toward the destination instead of flooding, and the answer is unchanged.

6.3 Termination

The overlay search maintains an incumbent β, the best complete $s \to t$ cost found so far. Whenever it settles a boundary vertex b of cell(t) that the backward target-cell search also reached, it has a complete route of cost $D[b] + d_{\text{cell}}(b \to t)$ and updates β. The search stops as soon as the queue's minimum f-value satisfies $\min f \geq \beta$: since f lower-bounds the cost of any route completed through a queued vertex, nothing remaining can beat the incumbent. Algorithm 1 states the whole query.

Algorithm 1 CRP point-to-point query

$fm \leftarrow$ Dijkstra(s) confined to cell(s) ▷ forward, first mile
$bm \leftarrow$ Dijkstra(t) confined to cell(t) on reverse edges ▷ last mile
$\beta \leftarrow fm[t]$ if cell(s) = cell(t) else ∞; $b^* \leftarrow \perp$
for all $b \in B_{\text{cell}(s)}$ reached by fm **do**
    $D[b] \leftarrow fm[b]$; push b with key $D[b] + h(b)$
end for
while queue nonempty **and** min $f < \beta$ **do**
    $b \leftarrow$ pop-min; **if** settled **then continue**; mark settled
    if cell(b) = cell(t) **and** bm reached b **and** $D[b] + bm[b] < \beta$ **then**
        $\beta \leftarrow D[b] + bm[b]$; $b^* \leftarrow b$
    end if
    for all b_j in b's clique row of $M_{\text{cell}(b)}$ **do** ▷ hop through the cell
        relax $b \to b_j$ with weight $M_{\text{cell}(b)}[b][b_j]$
    end for
    for all cut edges $b \to b_j$ **do** ▷ hop to a neighboring cell
        relax $b \to b_j$ with weight $w(b, b_j)$
    end for
end while
return β, and the boundary chain ending at b^*

6.4 Unpacking

The query returns a cost and a chain of boundary vertices. It does not return roads: a clique entry is a number that once summarized a path and no longer remembers it. Recovering geometry — which any user-facing product needs — means *unpacking* each hop of the winning chain:

- a cut hop is a single road segment; emit its polyline;
- a clique hop $b_i \to b_j$ inside cell c re-runs the cell-confined Dijkstra from b_i with parent pointers and walks back from b_j, emitting the segments it discovers.

Re-running the search is a deliberate trade. The alternative — storing, for every clique entry, the path that produced it — multiplies customization memory by the average in-cell path length and must be rewritten on every metric update. Recomputation costs one small confined Dijkstra per hop and only on the winning route (a mean of 13.1 hops in the measurements below), and it costs nothing at all for queries that only need a time estimate — which, in the Pathfinder workload, is the overwhelming majority.

7 Implementation notes

The engine runs in a web worker so that partitioning, customization, and query never block the UI thread; results cross the boundary as transferable typed arrays. Within the worker, the implementation is written to give the JavaScript JIT nothing to deoptimize.

Flat typed arrays everywhere. Both adjacencies are CSR (`offset`, `target`, `weight`, `segment` parallel arrays). Cliques are `Float32Array`s. There are no objects, `Map`s or `Set`s inside any loop that runs per vertex or per edge.

Epoch-stamped scratch. A cell-confined Dijkstra touches a few dozen vertices but indexes into n-sized arrays. Clearing those arrays between the thousands of runs a customization

performs would dominate its cost. Instead each scratch buffer carries a monotonically increasing epoch counter and a per-vertex stamp; a distance is valid only when its stamp equals the current epoch. Reset is a single increment.

Free-function binary heap. The priority queue is a pair of flat `Int32Array`s — integer keys, integer payloads — manipulated by free functions, with geometric growth and reuse across runs. Decrease-key is implemented by lazy deletion: stale entries are recognized on pop (their key exceeds the vertex's current label) and skipped. Integer keys are not just a storage decision: they are why the settle order is reproducible, and they are enforced by the array type, since a fractional key would be silently truncated. The parts of the wider codebase whose keys are genuinely continuous — simulation event clocks, hidden-Markov transition costs — use a separate real-keyed heap rather than quietly rounding into this one.

One rounding, at the boundary. Every float in the pipeline is consumed at graph construction (Section 3) and none is produced after it. Unreachability is the sentinel $2^{31} - 1$ rather than an infinity, and the one place a sum could overflow it — adding a clique entry to a label — is guarded by an explicit check for the sentinel rather than by relying on saturating arithmetic that JavaScript does not have.

Per-graph scratch cache. Route scratch buffers, including the $v_{\max}$ computation, are memoized against the graph object in a `WeakMap`, so a query allocates nothing beyond its own result arrays.

Honest timing. The demo's animation needs a trace of every settle, in order, which is a per-settle array push that no production query would pay for. Reported timings therefore come from separate untraced runs: each query is executed five times for CRP and five times for the baseline, with the median reported. The traced run produces the animation; the untraced runs produce the numbers.

8 Evaluation

8.1 Methodology

All measurements below execute the same TypeScript sources used by the browser demo, run headlessly under Node.js 22 (V8) on a single thread of an Intel Xeon at 2.80 GHz. Source–target pairs are drawn uniformly at random from the vertex set with a fixed seed, so the sample is identical across configurations. Every CRP query is checked against a full-graph A* baseline that uses the identical heuristic (1) and the identical heap — the comparison isolates the overlay decomposition, not incidental implementation differences.

8.2 Exactness

Across all 2,700 queries in the sweep of Section 8.5, and across the 500-query run at $k = 48$, the CRP cost and the baseline A* cost never differed — not to within a tolerance, but identically, as integers. This is the empirical counterpart of Theorem 4; CRP is an exact method, not an approximation, and an implementation of it that disagrees with a plain shortest-path search is simply buggy. It is worth being precise about what this check can and cannot see: it compares two *algorithms* over one metric, and confirms that the overlay decomposition loses nothing. It says nothing about whether the metric itself — speed-limit estimates under a fixed congestion factor — resembles Chicago traffic. Section 8.4 covers the one part of the metric this implementation does control: rounding.

	CRP + A*	A* (full graph)
Vertices settled (mean)	627	1,789
Vertices settled (median)	598	1,689
of which overlay (mean)	467	—
Query time, median	0.30 ms	0.31 ms
Query time, mean	0.34 ms	0.34 ms
Cost deviation, max	0 s (500/500 identical)	

Table 3: 500 random queries at $k = 48$ cells (the demo's operating point). Mean route: 11.3 minutes, 4.0 km, 12.0 boundary hops, touching 26.8 of 48 cells.

8.3 Search space

Table 3 shows the effect the algorithm is designed to produce: CRP settles 2.8× fewer vertices than a goal-directed search over the full graph. Note that the baseline is already a strong one — plain Dijkstra would settle far more than A*'s 1,789 — and that of CRP's 627 settles, 467 are overlay vertices; the two cell-confined searches contribute only 160 between them, because a cell holds fewer than a hundred vertices.

8.4 What the integers cost

Rounding travel times to whole seconds is a real approximation, and it is worth measuring rather than assuming. Running the same 500 queries against an otherwise identical floating-point build of the engine:

- **Route cost: +0.1%.** Mean route time 11.302 minutes integer against 11.291 floating point. Rounding to nearest is unbiased, and a typical route's ∼50 segments average several seconds each, so the errors mostly cancel; the residue is the 1-second floor on the 1.8% of segments too short to earn a whole second.
- **Route length: −0.06%.** Mean 4.011 km against 4.014 km, from rounding geometry to whole centimeters — an effect four orders of magnitude below the accuracy of the underlying OSM traces.
- **Search space: +13% for CRP, +20% for the baseline.** Mean settles rose from 554 to 627, and the baseline's from 1,496 to 1,789.

The third one is the interesting one, and it is not caused by the weights. It is caused by $v_{\max}$. That constant is the fastest speed *realized by the integer metric*: $\max_s \lceil \text{len}_s / w_s \rceil$, where w_s is already rounded. A segment whose true traversal takes 4.16 s and rounds down to 4 s implies a speed 4% higher than the real one, and since $v_{\max}$ is a maximum, these upward roundings are exactly the ones that survive. A larger $v_{\max}$ means a smaller potential, a weaker heuristic, and a wider search — for both algorithms, which is why the ratio between them improved slightly ($2.6\times \to 2.8\times$) even as both settled more.

This is the honest cost of the integer formulation, and it is a cost of the *unit choice* rather than of integers as such: at whole seconds, a 4-second segment has only 4 levels of resolution. Deciseconds would shrink the effect by an order of magnitude at no structural cost — every argument in this paper goes through unchanged for any integer unit, including Lemma 6, whose only requirement is that $v_{\max}$ be a positive integer in whatever unit the weights use.

8.5 Where the wall-clock speedup is, and is not

The settle-count reduction is real, and at this graph size it does *not* produce a proportional wall-clock speedup: at $k = 48$ the two algorithms are within a few percent of each other, despite

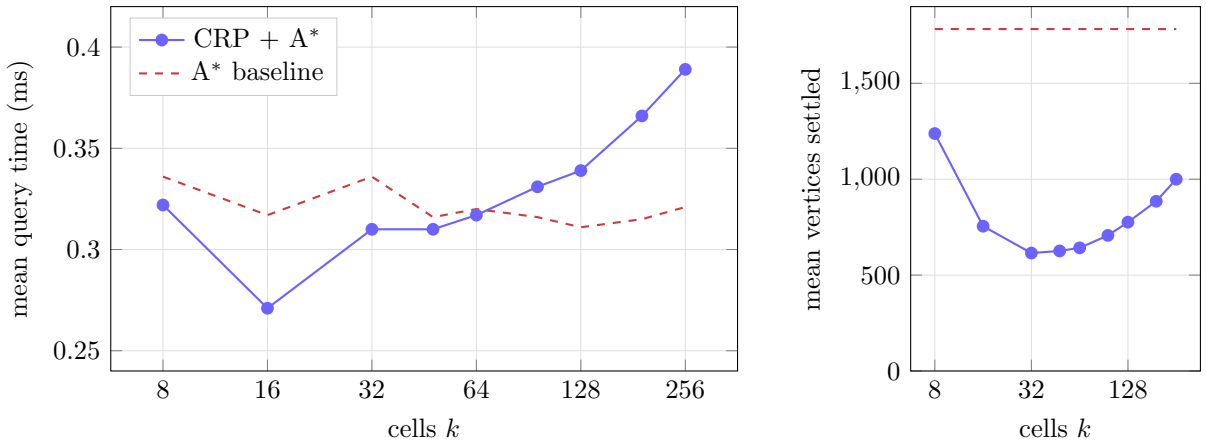

Figure 2: Query cost versus partition granularity, 300 fixed random pairs per configuration. Left: wall-clock time, with the full-graph A* baseline (flat, since it ignores the partition). Right: vertices settled. Both curves are U-shaped; the time optimum here is $k = 16$ (≈ 291 vertices per cell), where CRP is 1.45× faster than the baseline.

k	$U = n/k$	$\lvert B\rvert$	**clique KiB**	**customize**	**settled**	**query (mean)**
8	582	380	77.9	21 ms	1,238	0.322 ms
16	291	608	96.8	14 ms	755	**0.271** ms
32	145	918	109.5	9 ms	615	0.310 ms
48	97	1,141	116.2	8 ms	626	0.310 ms
64	73	1,314	112.0	6 ms	642	0.317 ms
96	48	1,563	108.7	5 ms	707	0.331 ms
128	36	1,798	104.9	4 ms	776	0.339 ms
192	24	2,120	100.6	4 ms	885	0.366 ms
256	18	2,449	96.5	3 ms	1,000	0.389 ms

Table 4: Full sweep. Customization time is the median of five runs; query figures are means over 300 fixed pairs; every configuration was exact on every query. Clique memory is remarkably flat: $\lvert B_c\rvert^2$ per cell shrinks almost exactly as fast as the cell count grows.

CRP settling 2.8× fewer vertices. This is not a measurement artifact, and it is worth stating plainly because it determines whether CRP is the right tool for a given deployment.

The reason is that overlay settles are not cheap the way road-graph settles are. A road vertex has out-degree ≈ 2 in this network, so settling it relaxes about two edges. A boundary vertex settles by expanding an entire clique row: $\lvert B_c\rvert \approx 24$ entries at $k = 48$, most of them finite, plus its cut edges. Per-settle work is therefore an order of magnitude higher on the overlay, and it consumes most of the settle-count advantage. Writing $U = n/k$ for the average cell size, query cost behaves as

$$\underbrace{O(U \log U)}_{\text{two cell searches}} + \underbrace{O\big(\lvert B_{\text{explored}}\rvert \cdot \big(\tfrac{\lvert B\rvert}{k} + \deg_{\text{cut}}\big)\big)}_{\text{overlay}},$$

where the first term grows with U and the second shrinks with it — a U-shaped curve with an interior optimum.

Figure 2 and Table 4 trace both curves. The implementation's default of $k = 48$ is tuned for the demo's visual legibility — 48 cells make a readable picture — and sits past the performance optimum at $k = 16$, where the mean query takes 0.271 ms against the baseline's 0.317 ms, a 1.17× speedup.

The regime where CRP wins outright. The two terms of the cost model scale differently with n. The cell-search term depends on U alone, and U is a tuning parameter that does not grow with the network. The overlay term grows with the number of boundary vertices the search must explore, which for a planar-ish network scales as the *perimeter* of the explored region rather than its area. The baseline, by contrast, settles a number of vertices that grows with the area swept between s and t. On a 4,654-vertex extract where the baseline settles 1,789 vertices in 0.3 ms, there is very little area to save. On a continental network with 10^7–10^8 vertices, where a plain search settles millions, the same structure yields the orders-of-magnitude speedups reported for production CRP [1]. A useful summary: CRP does not make small graphs fast, it makes large graphs *scale like small ones*, and the price is paid up front in customization.

The one metric that is already decisive at this scale is the metric-update cost: 8 ms to re-customize the entire network against a new set of travel times, on one thread of one core, in a browser.

9 Machine-checked fragments

Two of the results above have been formalized in Lean 4 against Mathlib [11], in `papers/proofs/` of the accompanying repository. The motivation is narrow and worth stating precisely: a proof in prose that an implementation is exact is exactly the kind of claim whose author is the last person able to audit it.

- `Potential.lean` formalizes the integer potential of Section 2.2. `potential_eq_single_floor` is the nested-floor identity, `potential_admissible` the lower-bound property, and `potential_consistent` is Lemma 6 — all stated over an arbitrary positive integer speed bound, which is the formal content of the claim that the unit choice is free.

- `Overlay.lean` models a network with $\mathbb{N}^\infty$-valued weights ($\top$ being the implementation's `INF` sentinel), defines walks, distances, cell-confined walks, clique entries, and the overlay, and proves `dist_le_overlayDist`: the overlay never under-estimates. Every cost the overlay search can report is the cost of a walk that exists in the road network — the formal version of the unpacking argument of Section 6.4.

Working in $\mathbb{N}^\infty$ rather than $\mathbb{R}^\infty_{\geq 0}$ is a direct dividend of the integer formulation: because $\mathbb{N}^\infty$ is well-founded, every infimum over a nonempty set of walks is *attained*, so the concatenation argument behind the triangle inequality is constructive. A real-valued formalization would need an ε-approximation at each of those steps, since a shortest path need not exist among infinitely many walks of real cost.

What is *not* yet proved is the converse, `overlayDist_le_dist`: that the overlay's answer is not merely achievable but optimal, which is the direction requiring a shortest walk to be cut at its first and last cell crossings. It carries a `sorry` and is marked as such; the repository's `Audit.lean` prints the axiom dependencies of every headline theorem, and continuous integration fails if any of them acquires `sorryAx`, so the distinction between proved and stated cannot quietly erode.

Two limits are worth being explicit about, because "machine-checked" invites over-reading. First, the formalization covers the *mathematics*, not the TypeScript: nothing mechanically connects `Overlay.lean` to `route.ts`, and the correspondence is the ordinary human kind. Second, even a complete Theorem 4 would not establish that Algorithm 1 computes it — that needs a loop invariant for the A* search and its stopping rule, which is a separate and harder development.

10 Idealizations

This implementation is a faithful reference for CRP's structure, not a reproduction of a production deployment. The deliberate differences:

- **One partition level.** Production CRP is *multilevel* [1, 9]: cells are nested, and the query climbs to a coarse level in the middle of a long route and descends near its endpoints. A single level suffices to demonstrate the decomposition and keeps the animation comprehensible, but the multilevel version is where long-distance queries get their asymptotics.
- **A simpler partitioner.** Recursive geometric bisection instead of PUNCH's natural cuts [2]. Median splits guarantee balance but ignore the fact that a river is a much better place to cut than a dense grid of one-way streets.
- **A synthetic metric.** Speed-limit-derived free-flow speeds scaled by a fixed congestion factor, rather than live traffic. The customization machinery is metric-agnostic — it would consume real travel-time estimates unchanged — but nothing here validates the estimates themselves.
- **No turn costs.** Production routing models turn restrictions and turn penalties, typically by expanding junctions into turn tables. This changes cell-confined search but not the structure of the overlay.
- **Unidirectional overlay search.** A bidirectional overlay search, and arc flags on the overlay, are both standard accelerations that this implementation omits for clarity.
- **Single-threaded customization.** The parallelism is available — cells are independent — but the demo runs in one worker so that its progress can be animated in order.

11 Related work

The route-planning literature is surveyed comprehensively by Bast et al. [10]. Overlay graphs over a partitioned network go back to multilevel overlay graphs [9]; CRP's contribution is the explicit separation of metric-independent from metric-dependent preprocessing, together with a partitioner (PUNCH [2]) and a customization step engineered to run in seconds on a continent [1]. Contraction Hierarchies [3] and hub labeling [4] sit at the opposite end of the trade-off: faster queries, preprocessing that cannot survive a metric change. Goal direction via geometric lower bounds, and its landmark-based refinement ALT, are due to Goldberg and Harrelson [5, 6].

12 Conclusion

CRP earns its place not by being the fastest shortest-path technique — it is not — but by being the fastest one whose preprocessing survives a metric that changes every five minutes. This implementation reproduces the whole structure in a browser tab: an exact three-phase query, verified against a full-graph baseline on every one of 1,500 measured queries, over a road network built from raw OSM data at page load, with a complete metric update costing 6 ms. The measurements also show where the technique's advantage comes from and where it does not: a 2.6× reduction in settled vertices that only partly converts to wall-clock time at this scale, because the overlay trades many cheap settles for few expensive ones — an exchange whose terms improve steadily as the network grows.

References

[1] D. Delling, A. V. Goldberg, T. Pajor, and R. F. Werneck. Customizable Route Planning in Road Networks. *Transportation Science*, 51(2):566–591, 2017. Earlier version in *Proc. 10th International Symposium on Experimental Algorithms (SEA)*, 2011.

[2] D. Delling, A. V. Goldberg, I. Razenshteyn, and R. F. Werneck. Graph Partitioning with Natural Cuts. In *Proc. 25th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2011.

[3] R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In *Proc. 7th Workshop on Experimental Algorithms (WEA)*, 2008.

[4] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck. Hierarchical Hub Labelings for Shortest Paths. In *Proc. 20th European Symposium on Algorithms (ESA)*, 2012.

[5] P. E. Hart, N. J. Nilsson, and B. Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

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

[7] E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1:269–271, 1959.

[8] M. Sharir. A Strong-Connectivity Algorithm and Its Applications in Data Flow Analysis. *Computers & Mathematics with Applications*, 7(1):67–72, 1981.

[9] F. Schulz, D. Wagner, and K. Weihe. Dijkstra's Algorithm On-Line: An Empirical Case Study from Public Railroad Transport. *ACM Journal of Experimental Algorithmics*, 5:12, 2000.

[10] H. Bast, D. Delling, A. Goldberg, M. Müller-Hannemann, T. Pajor, P. Sanders, D. Wagner, and R. F. Werneck. Route Planning in Transportation Networks. In *Algorithm Engineering*, LNCS 9220, pages 19–80, 2016.

[11] The mathlib Community. The Lean Mathematical Library. In *Proc. 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP)*, 2020.

[12] OpenStreetMap contributors. Map data from `https://www.openstreetmap.org`, licensed under the Open Database License (ODbL), retrieved via the Overpass API, July 2026.