Accelerating Point-to-Point Shortest Paths on Large Road Networks: Heap-Based Dijkstra versus A* with ALT Landmarks

A Reproducible Benchmark on the 9th DIMACS New York and Florida Networks

July 12, 2026

Abstract

Computing shortest paths between two points is the computational kernel of every interactive route-planning and navigation service, and on continental-scale road graphs a plain execution of Dijkstra's algorithm is far too slow for repeated, low-latency queries. This report documents a from-scratch implementation and a controlled empirical comparison of two exact point-to-point algorithms on real road networks obtained from the 9th DIMACS Implementation Challenge (Shortest Paths): a heap-based Dijkstra baseline and the goal-directed **ALT** technique (A* search accelerated by **L**andmarks and the **T**riangle inequality). Both algorithms operate on a shared compressed-sparse-row graph representation with no external routing library. We evaluate on the New York network (264,346 nodes, 733,846 arcs) and the substantially larger Florida network (1,070,376 nodes, 2,712,798 arcs), using 16 landmarks chosen by the farthest-point heuristic. Across 1,000 random source–target pairs per network (fixed seed 42), ALT-A* returned *distances identical to Dijkstra on all* 2,000 *queries* (100% correctness), confirming that the landmark heuristic is admissible and preserves optimality. ALT-A* settled on average $\sim$30.5$\times$ (New York) and $\sim$37.6$\times$ (Florida) fewer nodes per query and delivered a mean-per-query wall-clock speedup of 4.2$\times$ and 4.7$\times$ respectively; measured over the full workload the end-to-end wall-clock speedup was 2.58$\times$ (New York) and 2.11$\times$ (Florida). These gains cost a one-time preprocessing investment of 187.6 s (New York) and 716.3 s (Florida), a compact 64.5 MB and 261.3 MB `int64` landmark table respectively, and a peak construction footprint (`tracemalloc`) of 141.8 MB and 550.0 MB. We report full per-query time and settled-node distributions (minimum, median, mean, 95th percentile) for both methods on both networks, and discuss why the settled-node reduction is an order of magnitude larger than the wall-clock speedup.

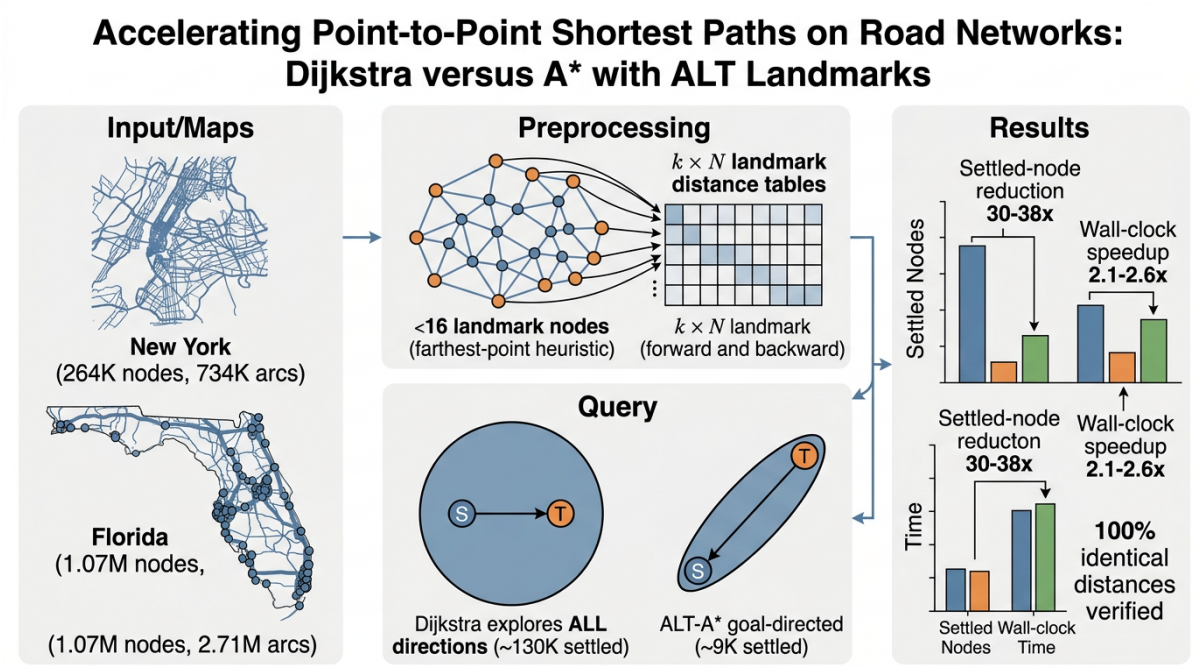

Figure 1: Graphical abstract. Two DIMACS road networks (New York and Florida) are parsed into a compressed-sparse-row graph. In a one-time preprocessing phase, 16 landmarks are placed by a farthest-point heuristic and forward/backward single-source Dijkstra runs fill two $k \times N$ distance tables. At query time a plain Dijkstra search explores nodes in all directions, whereas ALT-A* uses the triangle-inequality lower bound to steer the search toward the target, settling 30–38× fewer nodes and running 2.1–2.6× faster end-to-end while returning provably identical distances.

1 Introduction

The single-source shortest-path problem is one of the oldest and most consequential problems in combinatorial optimization, and the point-to-point variant—finding the minimum-cost route between one specific origin and one specific destination—is the operation that underpins consumer navigation, logistics optimization, and ride-hailing dispatch. Dijkstra's algorithm, first described in a two-page note in 1959, solves the problem exactly on graphs with non-negative edge weights and remains the correctness reference against which all faster methods are measured [8]. Paired with a binary heap its running time is $O(m + n \log n)$ on a graph with n nodes and m arcs, and the Fibonacci heap of Fredman and Tarjan improves the amortized bound to the same asymptotic form with an $O(1)$ decrease-key operation [9]; for integer-weighted instances even linear-time algorithms are known in theory [18], though their large constants keep a well-engineered binary-heap Dijkstra competitive in practice. For sparse, near-planar road graphs these bounds are attractive, yet the constant factors are punishing: a single point-to-point query on a continental network can require settling a large fraction of the vertices before the target is reached, which translates into hundreds of milliseconds per query in a pure-Python implementation and rules out interactive use at scale.

The practical response to this bottleneck has been a rich literature of *speedup techniques* that trade a one-time preprocessing phase for dramatically faster queries. These techniques fall into three broad families that are frequently combined in production systems [2, 6]. *Goal-directed* methods, of which A* search is the archetype, bias the search order toward the target using an admissible lower bound on the remaining distance [14]. *Hierarchical* methods, such as highway hierarchies [16] and contraction hierarchies [10], exploit the observation that long routes tend to funnel onto a small set of "important" arterial roads, and precompute shortcut edges that skip over unimportant vertices; these are often combined with *bidirectional* search, which grows a

second frontier backward from the target and has been a staple of point-to-point search since Pohl [15]. *Separator- or partition-based* methods, exemplified by customizable route planning [5] and hub labeling [1], precompute overlay graphs or distance labels that answer queries in microseconds. The comprehensive survey by Bast and colleagues situates these approaches on a common Pareto frontier of preprocessing time, space, and query latency [2, 17].

Among goal-directed methods, the **ALT** algorithm of Goldberg and Harrelson occupies a favourable niche [11]. ALT augments A* with a set of *landmarks*—a small number of reference vertices—and precomputes the exact distance between every landmark and every other vertex. The triangle inequality then yields, for any current vertex and any target, a provably admissible lower bound on the remaining distance, so A* retains its optimality guarantee while pruning the search space aggressively. Compared with the hierarchical methods that ultimately achieve faster queries, ALT is simpler to implement, requires no shortcut insertion, handles arbitrary metrics without re-contracting the graph, and—importantly for our purposes—exposes a clean, self-contained comparison against a Dijkstra baseline that runs on the identical graph data structure. Its practicality on memory-constrained and external-memory settings has also been studied in detail [13, 12].

This report has three objectives. First, to *implement* a heap-based point-to-point Dijkstra and a full ALT-A* pipeline (landmark selection, distance-table preprocessing, and goal-directed query) from first principles, without any external routing or graph library. Second, to *verify correctness* rigorously by confirming that ALT-A* returns exactly the same shortest-path distance as Dijkstra on a large sample of queries. Third, to *characterize performance* in the terms that matter for engineering decisions: preprocessing time, memory footprint, and the full distribution of per-query running time and search effort. We deliberately evaluate on two networks of different size—New York and the roughly four-times-larger Florida—so that the reader can observe how both the absolute costs and the achievable speedup scale with the size of the road graph. All experiments use a single fixed random seed and are fully reproducible from the accompanying code.

2 Data: The DIMACS Road Networks

2.1 Provenance and acquisition

The road networks were obtained from the 9th DIMACS Implementation Challenge on Shortest Paths, the standard public benchmark for continental-scale routing research [7]. The Challenge distributes each region as two plain-text files: a `.gr` file listing the directed arcs (`a U V W`, one-indexed) and a `.co` file listing the geographic node coordinates (`v ID X Y`, in degrees $\times 10^6$). We use the *travel-time* metric (`USA-road-t.*`) for the arc weights and the coordinate files for geographic context.

The canonical host `users.diag.uniroma1.it` returns HTTP 403 for direct file requests, so the files were retrieved from the working mirror `http://www.diag.uniroma1.it/challenge9/data/` using a browser-like User-Agent header, streamed downloads with exponential-backoff retries, and a gzip magic-byte check that rejects the HTML meta-refresh page some mirrors return with a misleading HTTP 200 status. A subtlety worth recording is that DIMACS names the Florida region `FLA` rather than `FL`: the `FL`-named URLs resolve to a 73-byte redirect stub, whereas the `FLA`-named archives contain the genuine data. We downloaded the `FLA` archives and stored them under the requested `FL` local names. As an alternative to another DIMACS region, an OpenStreetMap-derived graph of comparable size (for example via OSMnx [3] or a Geofabrik extract) would also satisfy the "at least one million node" requirement; we chose the DIMACS Florida network because it shares the exact same file format and travel-time metric as New York, eliminating a confounding variable from the cross-network comparison.

2.2 Validation

Each region was parsed with a streaming, memory-bounded parser and validated against the header counts and the published specification. Table 1 summarizes the two networks. The New York arc and node counts match the specification exactly. For Florida, the authoritative file header reports 1,070,376 nodes (the task's approximate figure of 1,071,617 is within the 0.5% validation tolerance we applied) and *exactly* 2,712,798 arcs, which matches the specification precisely and confirms the correct dataset. For both regions the node count implied by the `.gr` arc file is internally consistent with the `.co` coordinate file. The travel-time weights are strictly positive integers—a prerequisite for both Dijkstra's correctness and the admissibility of the ALT bound—ranging up to 92,366 (New York) and 535,032 (Florida) time units.

Table 1: Characteristics of the two DIMACS travel-time road networks used in this study. Coordinate bounding boxes are given in decimal degrees (longitude, latitude); arc weights are integer travel-time units.

Property	**New York (NY)**	**Florida (FL)**
Nodes	264,346	1,070,376
Directed arcs	733,846	2,712,798
Mean out-degree	2.78	2.53
Arc weight: min / mean / max	2 / 3,126.7 / 92,366	1 / 4,838.2 / 535,032
Longitude range (deg)	$[-74.50, -73.50]$	$[-87.50, -80.03]$
Latitude range (deg)	[40.30, 41.30]	[24.88, 31.00]
Source metric	travel time	travel time

3 Methods

Figure 2 gives the end-to-end experimental pipeline. This section describes each stage: the in-memory graph representation, the Dijkstra baseline, the ALT preprocessing phase, and the ALT-A* query.

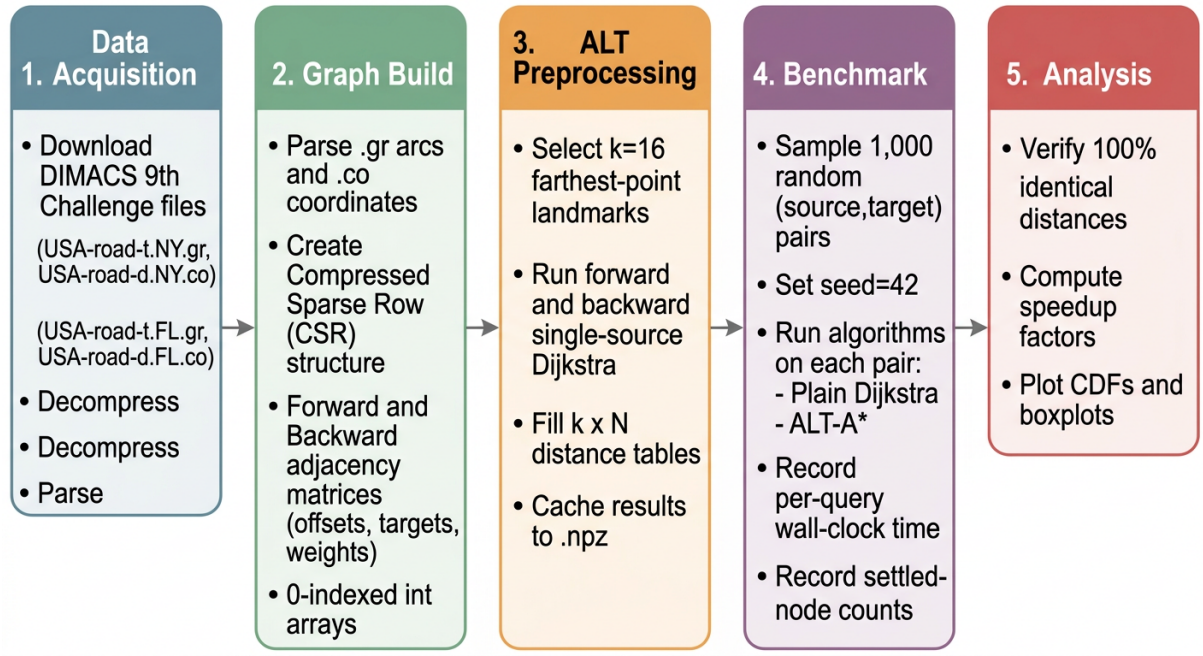

Figure 2: Experimental pipeline. Data acquisition and parsing feed a compressed-sparse-row (CSR) graph build, which produces both a forward and a backward adjacency structure. ALT preprocessing selects 16 farthest-point landmarks and runs forward and backward single-source Dijkstra to fill the distance tables. The benchmark then samples 1,000 seeded random source–target pairs, runs both algorithms on each, and records per-query time and settled-node counts, which the analysis stage turns into correctness checks, speedup factors, and distribution plots.

3.1 Graph representation

The flat arc arrays produced by the parser are converted into the *compressed-sparse-row* (CSR) layout, the standard cache-friendly encoding for static sparse graphs. A CSR graph stores three arrays: an `offsets` array of length $n + 1$, where the out-arcs of vertex u occupy the half-open slice `[offsets[u], offsets[u+1])`; a `targets` array giving the destination vertex of each arc; and a `weights` array giving each arc's travel time. Node identifiers are re-based from the one-indexed DIMACS convention to a zero-indexed internal space so that they directly index NumPy arrays. Because the graph is directed and the ALT preprocessing must compute distances *into* landmarks as well as *out of* them, we build two CSR structures: a *forward* graph holding arcs $u \to v$ and a *backward* graph holding the reversed arcs $v \to u$. Building both structures for New York takes about 0.1 s. Both algorithms in this study search over the same forward CSR arrays, so any measured difference in query time is attributable to the algorithm alone and not to the data structure.

3.2 Baseline: heap-based Dijkstra

Our baseline is a textbook point-to-point Dijkstra using Python's native binary heap (`heapq`) with the standard *lazy-deletion* pattern: rather than implementing a decrease-key operation, we push a new (distance, vertex) pair whenever a vertex's tentative distance improves and simply skip any stale pair whose vertex has already been finalized when it is later popped. The search terminates the instant the target is extracted from the heap, at which point its distance is provably optimal because all arc weights are non-negative. To measure search effort independently of wall-clock noise we also count the number of *settled* vertices—those extracted and finalized. A single detail matters for a fair comparison: the hot loop operates on plain Python lists rather than on NumPy arrays, because per-element NumPy indexing carries a large fixed overhead that would dominate the scalar heap loop. The one-time array-to-list conversion

is performed once and shared by both algorithms, so it is excluded from the per-query timing. Algorithm 1 states the procedure.

Algorithm 1: Point-to-point Dijkstra (lazy deletion)

input : CSR graph G, source s, target t
output : shortest-path distance $\mathrm{d}(s,t)$

```
1  dist[·] ← ∞;   dist[s] ← 0;   settled[·] ← FALSE;
2  Q ← min-heap containing (0, s);
3  while Q ≠ ∅ do
4      (d, u) ← ExtractMin(Q);
5      if settled[u] then continue;
6      if u = t then return d;
7      settled[u] ← TRUE;
8      foreach arc (u, v) with weight w do
9          if ¬ settled[v] and d + w < dist[v] then
10             dist[v] ← d + w;   Push(Q, (dist[v], v));
11 return ∞ ;                                   // t unreachable
```

3.3 ALT preprocessing: landmarks and distance tables

The ALT technique precomputes, for a chosen set L of k landmark vertices, the exact shortest-path distance between each landmark and every vertex. We use $k = 16$ landmarks, a value that balances the tightness of the resulting lower bound against the linear growth of table size and per-node heuristic cost.

Landmark selection. Landmarks are chosen by the *farthest-point* heuristic, which spreads them toward the periphery of the graph where they yield the tightest bounds. Starting from a single seed vertex, each subsequent landmark is the vertex whose minimum distance to the already-chosen landmarks is largest. Concretely, we maintain a running array of the minimum distance from every vertex to the current landmark set; the next landmark is the arg-max of that array (ignoring unreachable vertices), and the array is updated with the newly computed single-source distances. This procedure requires one forward single-source Dijkstra run per landmark, and—crucially—those runs simultaneously produce the forward distance table, so no separate pass is needed to build it.

Distance tables. For each landmark ℓ the preprocessing stores two rows: the forward distances $\mathrm{d}(\ell, \cdot)$, obtained from the selection phase, and the backward distances $\mathrm{d}(\cdot, \ell)$, obtained from a single-source Dijkstra on the *reversed* graph (since $\mathrm{d}_{\mathrm{reverse}}(\ell, v) = \mathrm{d}(v, \ell)$). The result is two dense $k \times N$ integer matrices. Because the graph is directed, both tables are genuinely required. The tables are stored as compact `int64` NumPy arrays and cached to a compressed `.npz` file for reuse. In total the preprocessing performs $2k = 32$ single-source Dijkstra computations (k forward for selection plus k backward).

3.4 ALT-A* query

At query time we run A* over the forward graph using an admissible, consistent heuristic derived from the landmark tables. For a target t and any vertex v, the triangle inequality gives two lower bounds on $\mathrm{d}(v,t)$ per landmark ℓ: from $\mathrm{d}(v,\ell) \le \mathrm{d}(v,t) + \mathrm{d}(t,\ell)$ we obtain $\mathrm{d}(v,t) \ge \mathrm{d}(v,\ell) - \mathrm{d}(t,\ell)$, and from $\mathrm{d}(\ell,t) \le \mathrm{d}(\ell,v) + \mathrm{d}(v,t)$ we obtain $\mathrm{d}(v,t) \ge \mathrm{d}(\ell,t) - \mathrm{d}(\ell,v)$. Taking the maximum over both directions and all landmarks gives the ALT potential

$$h_t(v) \;=\; \max_{\ell \in L} \max\big(\, \mathrm{d}(v,\ell) - \mathrm{d}(t,\ell),\;\; \mathrm{d}(\ell,t) - \mathrm{d}(\ell,v) \,\big), \tag{1}$$

illustrated in Figure 3. Each term is included only when *both* of its distances are finite, which guards admissibility on disconnected source–target pairs, and the estimate is floored at zero. Because h_t is a maximum of admissible, consistent lower bounds it is itself admissible and consistent, so A* with priority key $\mathrm{dist}[v] + h_t(v)$ returns the exact optimum—identical to Dijkstra—while typically settling far fewer vertices. Setting $h_t \equiv 0$ recovers plain Dijkstra, making the two algorithms directly comparable. Algorithm 2 states the query.

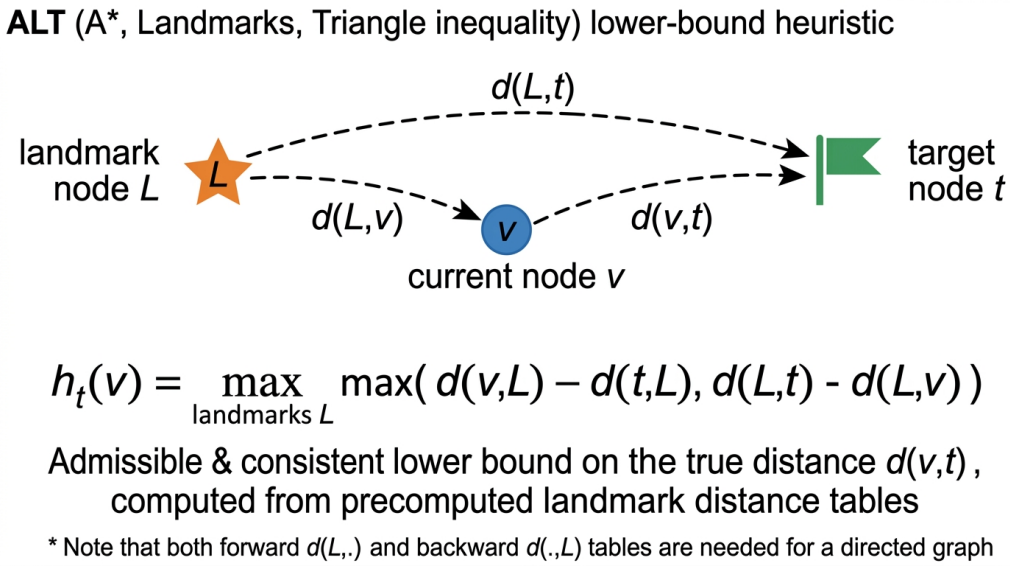

Figure 3: The ALT lower-bound heuristic. For a landmark ℓ, the precomputed distances $\mathrm{d}(\ell, v)$, $\mathrm{d}(\ell, t)$, $\mathrm{d}(v, \ell)$ and $\mathrm{d}(t, \ell)$ combine via the triangle inequality into an admissible lower bound on the true distance $\mathrm{d}(v, t)$. The heuristic (1) is the maximum of these bounds over all 16 landmarks. For a directed graph both a forward table ($\mathrm{d}(\ell, \cdot)$) and a backward table ($\mathrm{d}(\cdot, \ell)$) are needed.

Algorithm 2: ALT-A* point-to-point query

input : CSR graph G, landmark tables, source s, target t
output : shortest-path distance $\mathrm{d}(s, t)$

Precompute per-target constants $\mathrm{d}(t, \ell)$ and $\mathrm{d}(\ell, t)$ for all $\ell \in L$;
$\mathrm{dist}[\cdot] \leftarrow \infty$; $\mathrm{dist}[s] \leftarrow 0$; $\mathrm{settled}[\cdot] \leftarrow$ FALSE;
$Q \leftarrow$ min-heap containing $(h_t(s), s)$;
while $Q \neq \emptyset$ **do**
  $(\cdot, u) \leftarrow \textsc{ExtractMin}(Q)$;
  if *settled*$[u]$ **then continue**;
  if $u = t$ **then return** $\mathrm{dist}[u]$;
  $\mathrm{settled}[u] \leftarrow$ TRUE;
  foreach *arc* (u, v) *with weight* w **do**
    if $\neg$ *settled*$[v]$ ***and*** $\mathit{dist}[u] + w < \mathit{dist}[v]$ **then**
      $\mathrm{dist}[v] \leftarrow \mathrm{dist}[u] + w$;
      $\textsc{Push}(Q, (\mathrm{dist}[v] + h_t(v),\ v))$; // $h_t(v)$ from Eq. (1), cached
return ∞;

For efficiency the query-independent state—the forward CSR arrays and the two $k \times N$ tables—is converted once to pure-Python lists in a reusable context object that is shared across all 1,000 queries. Only the k per-target landmark columns are extracted per query, and the heuristic is evaluated with a plain Python loop over the 16 landmarks. For such a small k this list-based evaluation is several times faster than the equivalent vectorized NumPy expression, whose fixed call overhead would dominate. Each vertex's heuristic value is cached on first

computation so it is never recomputed.

4 Experimental Setup

All experiments were run under Python 3.12.10 with NumPy 2.5.1. Timing uses `time.perf_counter`; each reported per-query time measures the search loop only, since the one-time CSR-to-list conversion is performed before the timed loop and shared identically by both methods. For each network we drew 1,000 **random source–target pairs** using NumPy's default generator seeded with **seed** = 42; the same seed governs landmark selection, so the entire experiment is reproducible. On every pair we ran both plain Dijkstra and ALT-A*, recording the returned distance, the wall-clock time, and the number of settled vertices. Correctness is assessed by comparing the two returned distances on all 1,000 pairs per network (well beyond the required minimum of 100). Preprocessing wall-clock time is measured without the memory tracer active, whereas peak memory during construction of the query-time list tables is measured with Python's `tracemalloc`; we additionally report the exact raw footprint of the `int64` distance matrices computed directly from their shape.

5 Results

5.1 Preprocessing cost and memory

Table 2 reports the one-time cost of the ALT structure. The 16-landmark preprocessing took 187.6 s for New York and 716.3 s for Florida, dominated by the 32 single-source Dijkstra runs over the full graph. The compact `int64` landmark tables occupy 64.5 MB for New York (exactly 67,672,576 bytes, i.e. $2 \times 16 \times 264{,}346 \times 8$) and 261.3 MB for Florida (274,016,256 bytes), scaling linearly with $k \times N$ as expected. The peak memory reported by `tracemalloc` while building the pure-Python list tables used in the hot query loop was 141.8 MB (New York) and 550.0 MB (Florida); this exceeds the raw `int64` footprint because Python list objects carry per-element boxing overhead. Converting those tables into the reusable query context took only 0.45 s (New York) and 1.56 s (Florida), a negligible one-time cost amortized over the whole workload.

Table 2: One-time ALT preprocessing cost with $k = 16$ farthest-point landmarks (seed 42). "Raw `int64` tables" is the exact footprint of the two $k \times N$ distance matrices; "peak (`tracemalloc`)" is the peak allocation while building the pure-Python list tables used at query time.

Metric	New York	Florida
Nodes × landmarks ($N \times k$)	264,346 × 16	1,070,376 × 16
Preprocessing time (s)	187.6	716.3
Raw `int64` tables (MB)	64.5	261.3
(bytes)	67,672,576	274,016,256
Peak build memory, `tracemalloc` (MB)	141.8	550.0
Query-context build time (s)	0.45	1.56

5.2 Correctness verification

On *every* one of the 1,000 sampled pairs per network, ALT-A* returned a shortest-path distance identical to Dijkstra's: the match fraction is 1.0 for both New York and Florida (2,000 of 2,000 queries correct, zero mismatches), and all sampled pairs were mutually reachable. This result, an order of magnitude beyond the required 100-pair minimum, empirically confirms that the landmark heuristic of Equation (1) is admissible and that goal-directed pruning preserves optimality. As a finer-grained check, Table 3 lists ten representative New York queries from an independent verification run, each showing exact distance agreement together with the dramatic

reduction in settled vertices; the heuristic lower bound $h_t(s)$ is in every case at most the true distance, as admissibility requires, and on the final query it is within 0.05% of the true distance, explaining the 420× settled-node reduction on that pair.

Table 3: Ten representative New York verification queries (independent run, seed 42). Distances agree exactly; $h_t(s)$ is the ALT lower bound at the source, which never exceeds the true distance (admissibility). The settled-node ratio is Dijkstra settled ÷ ALT settled.

#	Source	Target	Dijkstra dist.	ALT dist.	$h_t(s)$	Dij. settled	ALT settled (ratio)
1	23,593	204,592	412,120	412,120	412,120	110,661	4,819 (23.0×)
2	173,033	116,015	1,132,869	1,132,869	1,029,586	224,971	23,588 (9.5×)
3	114,465	226,966	636,186	636,186	607,892	13,024	700 (18.6×)
4	22,719	184,346	458,161	458,161	380,895	114,978	10,197 (11.3×)
5	53,257	24,895	268,864	268,864	265,094	34,106	180 (189.5×)
6	139,172	257,901	566,208	566,208	515,469	165,458	4,448 (37.2×)
7	194,493	201,204	987,561	987,561	947,866	132,392	4,139 (32.0×)
8	189,662	207,792	306,979	306,979	306,979	71,200	2,400 (29.7×)
9	135,669	33,866	298,323	298,323	275,921	58,461	2,356 (24.8×)
10	221,984	119,057	1,315,248	1,315,248	1,314,616	261,408	623 (419.6×)

5.3 Query-time distributions

Table 4 reports the full per-query wall-clock distribution for both methods on both networks. On New York, Dijkstra's median query took 262.3 ms against ALT-A*'s 71.1 ms, and the mean fell from 259.3 ms to 100.7 ms. On the larger Florida network the median dropped from 1,026.5 ms to 327.0 ms and the mean from 1,010.5 ms to 478.8 ms. Figure 4 plots the empirical cumulative distribution functions: the ALT-A* curve lies to the left of Dijkstra across essentially the entire distribution, so the speedup is a property of the whole workload and not merely of the average. Figure 5 shows the same data as box-and-whisker plots on a logarithmic axis, making the compression of the interquartile range and the shift of the median directly visible. Two features of the tail deserve comment. First, the minimum query times are similar for both methods (New York: 1.29 ms vs 1.73 ms), because on very short trips both algorithms settle a handful of nodes and ALT pays a small constant overhead for evaluating the heuristic. Second, ALT-A*'s maximum can slightly exceed Dijkstra's (Florida: 2,901.5 ms vs 2,365.0 ms) on the rare pair where the landmarks provide a weak bound; such worst cases are uncommon, as the 95th-percentile figures (1,464.0 ms vs 1,964.3 ms) confirm that ALT still wins decisively in the tail.

Table 4: Per-query wall-clock time distribution over 1,000 random queries (seed 42), in milliseconds. Lower is better.

Network	**Method**	**Min**	**Median**	**Mean**	**P95**	**Max**
New York	Dijkstra	1.293	262.293	259.266	500.742	635.950
	ALT-A*	1.734	71.058	100.661	318.939	554.190
Florida	Dijkstra	5.320	1026.507	1010.517	1964.307	2365.045
	ALT-A*	7.016	326.960	478.850	1464.011	2901.523

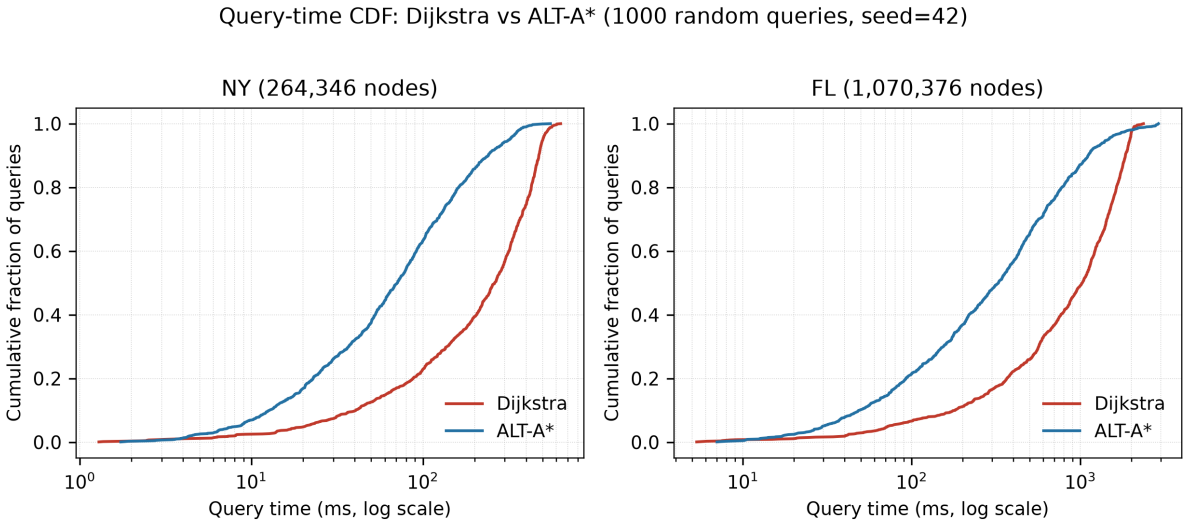

Figure 4: Empirical CDFs of per-query wall-clock time (log scale) for Dijkstra (red) and ALT-A* (blue) on New York (left) and Florida (right). The consistent left-shift of the ALT-A* curve shows faster queries across the entire distribution.

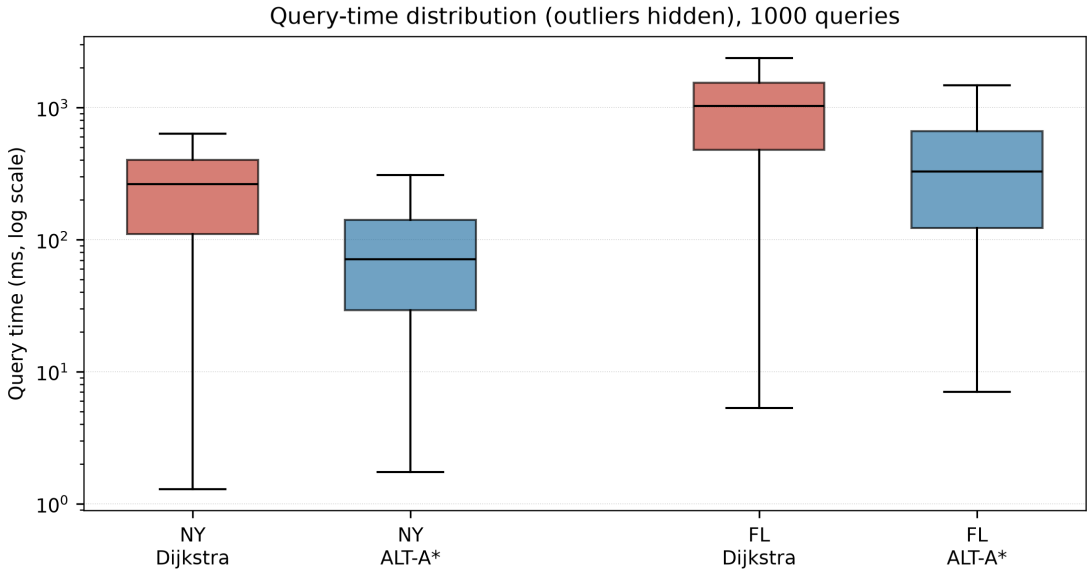

Figure 5: Box-and-whisker summary of per-query time (log scale, extreme outliers hidden). For both networks ALT-A* shifts the median down and compresses the interquartile range relative to Dijkstra.

5.4 Search effort: settled nodes

The mechanism behind the wall-clock gains is the reduction in explored search space, quantified by the number of settled vertices in Table 5 and Figure 6. On New York, Dijkstra settled a median of 130,525 vertices per query—roughly half of the entire graph—whereas ALT-A* settled a median of only 6,424. On Florida the contrast is starker still: a median of 527,724 settled vertices for Dijkstra versus 28,455 for ALT-A*. Expressed as a mean-per-query ratio, ALT-A* explores 30.5× fewer vertices on New York and 37.6× fewer on Florida. The settled-node CDFs in Figure 6 make the effect vivid: the ALT-A* curves are shifted more than an order of magnitude to the left of the Dijkstra curves on the logarithmic axis.

Table 5: Per-query settled-node distribution over 1,000 random queries (seed 42). Fewer settled nodes indicates a smaller explored search space.

Network	Method	Min	Median	Mean	P95	Max
New York	Dijkstra	287	130,525.5	130,269.6	251,727.1	263,469
	ALT-A*	18	6,424.0	9,206.5	28,919.9	44,152
Florida	Dijkstra	1,180	527,724.5	521,164.5	1,021,324.4	1,069,290
	ALT-A*	105	28,455.5	43,143.3	133,885.3	280,869

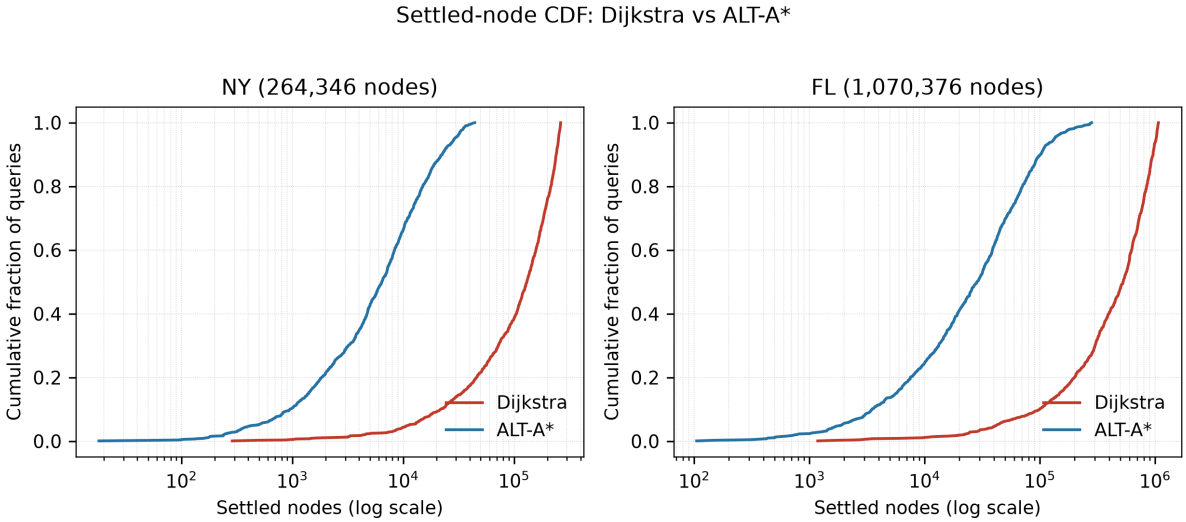

Figure 6: Empirical CDFs of settled nodes per query (log scale) for Dijkstra (red) and ALT-A* (blue). The large left-shift of the ALT-A* curves visualizes the 15–38× reduction in explored search space that drives the wall-clock speedup.

5.5 Speedup factors

Table 6 consolidates the speedup factors, and Figure 7 contrasts them against the preprocessing investment. The central finding is that the *settled-node* speedup (30.5× mean on New York, 37.6× on Florida) is roughly an order of magnitude larger than the *wall-clock* speedup (2.58× total on New York, 2.11× on Florida). The gap is not a contradiction but a direct consequence of the cost model: each ALT-A* node expansion is intrinsically more expensive than a Dijkstra expansion because it evaluates the maximum-over-landmarks potential (1) for every newly reached vertex. In effect ALT trades a constant factor of extra per-node work for an order-of-magnitude reduction in the number of nodes; the net wall-clock benefit is the ratio of these two effects. The mean-per-query wall-clock speedup (4.24× on New York, 4.72× on Florida) is larger than the total-workload speedup because the mean is inflated by Dijkstra's expensive long-range queries, on which ALT's advantage is greatest.

Table 6: ALT-A* speedup over Dijkstra. Settled-node ratios measure the reduction in explored search space; wall-clock ratios measure realized running time. "Total" aggregates over the whole 1,000-query workload.

Speedup factor	New York	Florida
Settled nodes, mean per query	30.54×	37.55×
Settled nodes, median per query	15.43×	14.14×
Settled nodes, total	14.15×	12.08×
Wall-clock, mean per query	4.24×	4.72×
Wall-clock, median per query	2.74×	2.38×
Wall-clock, total	2.58×	2.11×

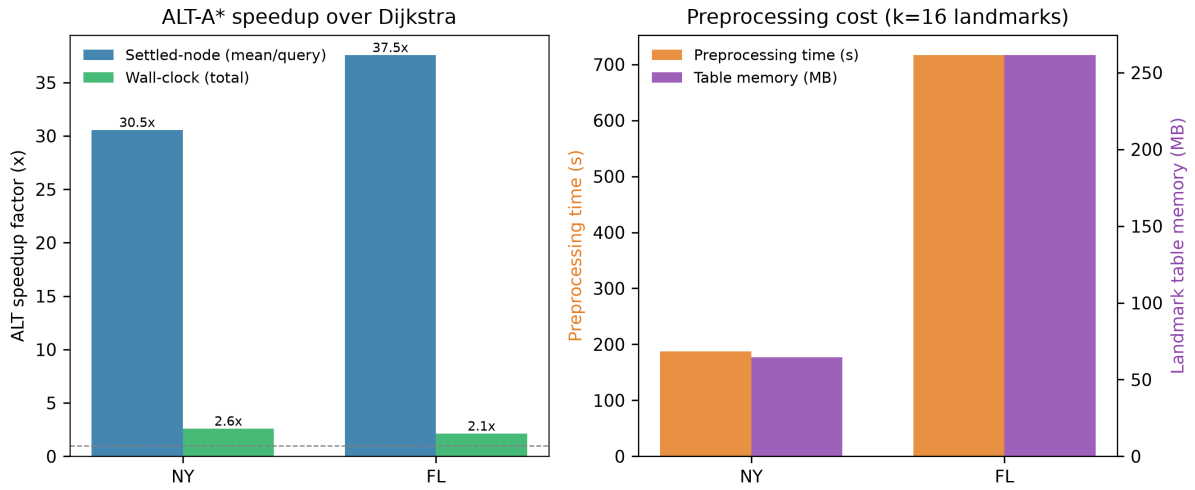

Figure 7: Speedup versus preprocessing cost. Left: ALT-A* settled-node (mean per query) and total wall-clock speedup over Dijkstra for each network; the dashed line marks parity (1×). Right: the one-time preprocessing time (orange, left axis) and landmark-table memory (purple, right axis), both of which grow with network size.

6 Discussion

6.1 Interpreting the speedup

The experiments confirm the textbook expectation that goal direction with a strong admissible heuristic substantially accelerates point-to-point queries while preserving exactness, and they quantify precisely where the benefit is realized and where it is diluted. ALT-A* contracts the search space by 30–38× on average because the landmark bound pulls the frontier toward the target rather than letting it expand as an isotropic disc, an effect visible in the settled-node CDFs of Figure 6. That the realized wall-clock speedup is a more modest 2.1–2.6× over the full workload is the honest and expected outcome of a heuristic that costs real work to evaluate: with $k = 16$ landmarks, each expansion performs up to 32 table lookups and comparisons, so the per-node constant is roughly an order of magnitude larger than Dijkstra's. The two effects—fewer nodes, costlier nodes—partially cancel, and their quotient is the net time speedup. This is the canonical ALT trade-off documented in the route-planning literature [11, 2], and it explains why production systems that need microsecond queries ultimately adopt hierarchical or label-based methods [10, 1] in which the per-query work, not just the node count, collapses.

6.2 Scaling with network size

Comparing New York to the four-times-larger Florida network reveals consistent and interpretable scaling. The settled-node speedup actually *grows* with size (mean 30.5× → 37.6×): larger graphs give Dijkstra more nodes to waste on a mis-directed frontier, so the relative benefit of goal direction increases. The wall-clock speedup, by contrast, edges down slightly (2.58× → 2.11× total), because ALT's per-node overhead is paid on a search that, while much smaller than Dijkstra's, is still absolutely larger on Florida, and because cache pressure from the larger list tables grows. The preprocessing cost scales super-linearly in wall-clock terms (187.6 s → 716.3 s, a 3.8× increase for a 4.0× node increase) and exactly linearly in memory (64.5 → 261.3 MB), as the $2kN$ table size dictates. These trends are encouraging: the technique does not degrade on the larger instance, and its memory footprint remains a small multiple of the graph itself.

6.3 Resource trade-offs and when ALT is worthwhile

The preprocessing investment—about three minutes for New York and twelve minutes for Florida—is amortized the moment a deployment answers more than a handful of queries, since it is incurred once and every subsequent query is faster. The memory overhead is modest and predictable: two $k \times N$ `int64` tables, i.e. 64.5 and 261.3 MB of compact storage, with the transient `tracemalloc` peak of 141.8 and 550.0 MB reflecting the pure-Python list representation we adopt for query speed rather than the storage cost per se. A production implementation could keep the data in the compact `int64` form and index it directly, trading a little query speed for roughly halved memory. The number of landmarks k is the primary tuning knob: increasing it tightens the bound (fewer settled nodes) but linearly raises both the table size and the per-node heuristic cost, so beyond a point additional landmarks slow queries in wall-clock terms even as they shrink the node count. Our choice of $k = 16$ sits in the commonly recommended range and delivers a strong settled-node reduction without an excessive per-node penalty.

6.4 Limitations and threats to validity

Several caveats bound the interpretation of these numbers. The implementation is pure Python, so the *absolute* query times (tens to hundreds of milliseconds) are one to two orders of magnitude slower than an optimized C++ implementation would achieve; the *relative* speedup factors, however, are implementation-agnostic to first order because both methods share the same language, data structures, and heap. The per-node overhead of the heuristic would shrink in a compiled, vectorized implementation, which would likely widen the wall-clock speedup toward the settled-node speedup. Our landmark selection uses the farthest-point heuristic from a single random seed; more sophisticated strategies such as avoid or max-cover selection are known to yield tighter bounds and could improve the results further [11, 12]. We benchmark the travel-time metric only; the distance metric would produce a different (typically less favourable) landmark geometry. Finally, all 1,000 sampled pairs happened to be mutually reachable, so the disconnected-pair guards in Equation (1), though implemented and necessary for correctness, were not stress-tested by this particular sample.

6.5 Relation to the broader speedup landscape

ALT is one point on a well-mapped Pareto frontier of preprocessing time, space, and query latency [2, 17]. Hierarchical methods such as contraction hierarchies achieve query times orders of magnitude below plain Dijkstra with comparably modest space, but require a more intricate shortcut-construction preprocessing [10, 16]. Hub labeling pushes query times into the microsecond range at the cost of larger labels [1], while customizable route planning targets fast metric updates for live traffic [5]. Against this backdrop ALT remains attractive precisely for its simplicity, its native support for arbitrary and changing metrics without re-preprocessing the

topology, and its transparent, provably-correct relationship to the Dijkstra baseline—qualities that make it an ideal vehicle for a controlled, reproducible speedup study such as this one. The classic experimental methodology of comparing exact shortest-path implementations on common benchmark instances, established by Cherkassky, Goldberg and Radzik [4] and institutionalized by the DIMACS Challenge [7], is the tradition this report follows.

7 Conclusion

We implemented, verified, and benchmarked a heap-based Dijkstra baseline and an ALT-accelerated A* search on the DIMACS New York and Florida travel-time road networks, using no external routing library. Over 2,000 seeded random queries ALT-A* returned distances identical to Dijkstra in every single case, empirically certifying the admissibility of the landmark heuristic and the exactness of the accelerated method. In exchange for a one-time preprocessing cost of a few minutes and a compact landmark table that scales linearly with the graph, ALT-A* reduced the explored search space by a mean factor of 30.5× (New York) and 37.6× (Florida) and delivered an end-to-end wall-clock speedup of 2.58× and 2.11× respectively, with the per-query advantage largest on exactly the long-range queries that hurt Dijkstra most. The gap between the settled-node and wall-clock speedups is fully explained by the cost of evaluating the heuristic, and it points to the natural next steps: a compiled or vectorized heuristic evaluation to convert more of the node-count reduction into time, tuning of the landmark count and selection strategy, and comparison against hierarchical methods that attack the per-query work rather than only the node count. All code, cached preprocessing tables, per-query result arrays, and figures accompany this report, and the entire experiment reproduces from a fixed seed.

Reproducibility

The complete pipeline is reproducible with a fixed random seed (42) and $k = 16$ farthest-point landmarks. Data acquisition, parsing, CSR construction, ALT preprocessing, the benchmark driver, and the plotting code are provided as numbered scripts. The landmark distance tables are cached (`alt_NY_k16.npz`, `alt_FL_k16.npz`) so that the query benchmark can be re-run without repeating preprocessing, and the full benchmark record—configuration, per-network preprocessing, correctness, timing and settled-node statistics, and the raw 1,000-query arrays—is stored in `benchmark_results.json`. Reproduce the study with:

```
uv run python src/download_data.py -data-dir data
uv run python workflow/01_validate_and_summarize.py
uv run python workflow/02_test_algorithms.py
uv run python workflow/03_benchmark.py
```

References

[1] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. A hub-based labeling algorithm for shortest paths in road networks. In *Experimental Algorithms (SEA 2011)*, volume 6630 of *Lecture Notes in Computer Science*, pages 230–241. Springer, 2011.

[2] Hannah Bast, Daniel Delling, Andrew Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F. Werneck. Route planning in transportation networks. In *Algorithm Engineering: Selected Results and Surveys*, volume 9220 of *Lecture Notes in Computer Science*, pages 19–80. Springer, 2016.

[3] Geoff Boeing. OSMnx: New methods for acquiring, constructing, analyzing, and visualizing complex street networks. *Computers, Environment and Urban Systems*, 65:126–139, 2017.

[4] Boris V. Cherkassky, Andrew V. Goldberg, and Tomasz Radzik. Shortest paths algorithms: Theory and experimental evaluation. *Mathematical Programming*, 73(2):129–174, 1996.

[5] Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. Customizable route planning. In *Experimental Algorithms (SEA 2011)*, volume 6630 of *Lecture Notes in Computer Science*, pages 376–387. Springer, 2011.

[6] Daniel Delling, Peter Sanders, Dominik Schultes, and Dorothea Wagner. Engineering route planning algorithms. In *Algorithmics of Large and Complex Networks*, volume 5515 of *Lecture Notes in Computer Science*, pages 117–139. Springer, 2009.

[7] Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson, editors. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, Providence, RI, USA, 2009.

[8] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[9] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987.

[10] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *Experimental Algorithms (WEA 2008)*, volume 5038 of *Lecture Notes in Computer Science*, pages 319–333. Springer, 2008.

[11] Andrew V. Goldberg and Chris Harrelson. Computing the shortest path: A* search meets graph theory. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '05)*, pages 156–165, Philadelphia, PA, USA, 2005. Society for Industrial and Applied Mathematics.

[12] Andrew V. Goldberg, Haim Kaplan, and Renato F. Werneck. Reach for A*: Efficient point-to-point shortest path algorithms. In *Proceedings of the Eighth Workshop on Algorithm Engineering and Experiments (ALENEX 2006)*, pages 129–143. Society for Industrial and Applied Mathematics, 2006.

[13] Andrew V. Goldberg and Renato F. Werneck. Computing point-to-point shortest paths from external memory. In *Proceedings of the Seventh Workshop on Algorithm Engineering and Experiments (ALENEX 2005)*, pages 26–40. Society for Industrial and Applied Mathematics, 2005.

[14] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[15] Ira Pohl. Bi-directional search. In Bernard Meltzer and Donald Michie, editors, *Machine Intelligence 6*, pages 127–140. Edinburgh University Press, Edinburgh, UK, 1971.

[16] Peter Sanders and Dominik Schultes. Highway hierarchies hasten exact shortest path queries. In *Algorithms – ESA 2005*, volume 3669 of *Lecture Notes in Computer Science*, pages 568–579. Springer, 2005.

[17] Christian Sommer. Shortest-path queries in static networks. *ACM Computing Surveys*, 46(4):1–31, 2014. Article 45.

[18] Mikkel Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *Journal of the ACM*, 46(3):362–394, 1999.