Divide & Conquer Street-Routing — Method & Experiments 1–3

Version: Oct 2025
Codebase anchors: `divideconquer.py`, `experiment1.py`, `experiment2.py`, `experiment3.py`

---

1) Executive summary

- **Goal.** Build practical city routes that follow *real streets*, starting at a depot and ending at another, while covering many delivery points.
- **Core idea (Divide & Conquer).**
    1. **Divide** the delivery set with a **quad-tree** into small, spatially compact regions.
    2. **Conquer** each region via a **greedy nearest-neighbor** mini-tour on the street network (no straight-line cheats).
    3. **Combine** regions into one city-wide route by **stitching shortest street paths** between region entries and exits.
- **Experiment 1 — Clean S→N path.** Corridor-aware; selects only regions near the shortest S→N street path; visits regions south→north.
- **Experiment 2 — Multi-depot.** Repeats the clean-path method across several start→end depot pairs and compares runtime/coverage/memory.
- **Experiment 3 — Priority-only (no corridor).** Picks **top-scoring regions** by delivery **priority/size**, independent of corridor distance; still orders them south→north to keep a clean narrative.

---

2) Method (what the router actually does)

2.1 Inputs

- **Street graph** `G` (OSMnx/NetworkX), nodes have lon/lat; edges carry length (meters).
- **Delivery points** CSV with columns: `lat, lon, priority` (1=high, 2=medium, 3=low). They are snapped to nearest graph node.
- **Depots** (`S, N, E, W, C`) with fixed lat/lon.

2.2 Divide — Quad-tree

- Recursively split the bounding box of deliveries into 4 quadrants until each region has ≤ *M* points (e.g., 15–20) or `max_depth` is reached.
- For each region store: points subset, centroid, bounds rectangle, depth, size.

2.3 Conquer — Regional mini-tour (street-aware)

- For a chosen region, pick an **entry node** (closest by **graph distance**) to the current position using a single-source shortest path (Dijkstra) from the current node.
- Inside the region, run **greedy nearest-neighbor** on **graph distances** to visit all deliveries. Expand the ordered nodes into a **street polyline** using `nx.shortest_path` between consecutive nodes.

2.4 Combine — Stitch regions into one route

- Connect: previous region's last node → next region's entry node via a **street shortest path**; repeat, then end at the terminal depot.

Guarantee: Every hop is a path embedded in the street graph, so the final polyline is *non-jumpy* and map-clean. (If NetworkX fails for rare pairs, the code falls back to a direct [u,v] edge; consider handling that carefully in future work.)

3) Experiments

3.1 Experiment 1 — Clean Single-Path (South → North)

Intent. Create a visually clean route that largely hugs the *shortest S→N street path* (the "spine"), visiting only regions that lie close to it.

Steps.

1. Compute the shortest S→N path on streets (the spine).
2. Build the quad-tree over all deliveries.
3. **Filter** regions whose centroids lie within a corridor band (e.g., 3 km) of the spine.
4. **Order** the kept regions by their projection along the spine (south→north).
5. For each region in order: connect to its entry; do regional NN tour; stitch back to the sequence.
6. Connect last region to **N**.

Outputs.

- Route PNG per `n` (visited/unvisited shown), quad-tree PNG, and a scalability panel plotting time/regions/coverage/efficiency.

Notes. No budget enforcement is applied in this clean-path variant; distance is observed/reported, not constrained.

3.2 Experiment 2 — Multi-Depot Comparison

Intent. Compare performance and coverage across several start→end depot pairs using the *same clean-path method*.

Pairs. Example: (S→E), (C→E), (W→S), (E→W), (N→S).

Steps. For each pair: load the same delivery set, run clean-path routing, measure computation time, memory (approx. peak), distance, coverage.

Outputs.

- **Heatmap/table** comparing distance, time, memory, and deliveries covered per depot pair.
- **Bar chart** of number of deliveries covered per pair.
- Region-distribution chart (nearest-depot counts) for intuition.

Notes. Also uses no budget; it's a head-to-head profile of geometry and workload differences by origin/destination.

3.3 Experiment 3 — Priority-Only Region Selection (No Corridor)

Intent. Make **priority** drive which regions are visited, not proximity to the S→N spine. Still travel S→N for readability.

Selection.

- Score each region by **priority weight**: `50×#(P1) + 30×#(P2) + 20×#(P3).`
- Filter out small regions: keep only those with ≥ *min_points_per_region*.
- Keep top-K regions by *(score, size)*; **no corridor distance** considered.
- Order selected regions south→north by spine projection; visit all points in each (regional NN) and stitch with street paths.

Visuals.

- **Priority-colored scatter** of all points (P1=red, P2=orange, P3=green).
- **Selected region overlays** (rectangles + centroids) on the same map.
- Standard route plot (visited vs unvisited) for each config.

Knobs. `min_points_per_region`, `top_k_regions`, quad-tree `max_points_per_region`.

4) Time & space complexity (practical big-O)

Let |V|=#nodes, |E|=#edges of the street graph; N=deliveries; R=regions; P_i=points in region i (ΣP_i=N).

Preprocessing

- KD-tree build for node snapping: `O(|V| log |V|)` (once).
- Shortest S→N spine: `O(|E| log |V|)` (once).
- Quad-tree: ~`O(N log N)`.

Per-region entry selection

- Use **one** single-source Dijkstra from the current node: `O(|E| log |V|)`; then take `argmin` over region nodes (constant factor per region).

Intra-region NN

- For P_i points: P_i iterations, each performs one Dijkstra from the current node ⇒ `O(Pᵢ · |E| log |V|)` per region; summed: `O(N · |E| log |V|)`.

Stitching between regions

- A handful of shortest paths between consecutive regions and to the end depot ⇒ `O(R · |E| log |V|)`.

Total time

- Dominated by many SSSPs: **`≈ O(N · |E| log |V|)`**.

Space

- Graph & KD-tree: `O(|V| + |E|)`.
- Deliveries/regions: `O(N + R)`.
- **Dijkstra cache** (LRU of single-source distance dicts): worst-case large; each cached entry stores distances to many nodes. Keep `maxsize` moderate and monitor resident memory.

Empirical tip. Runtime grows mainly with *(#regions visited)* × *(avg region size)* and the density of the street graph. The single-Dijkstra entry selection keeps constants much lower than running Dijkstra per candidate.

5) Plot & file glossary (what each artifact means)

Artifact / Pattern	Produced by	What it shows	How to read	Notes/Caveats
`Quad_Tree_Decomposition_Size_{n}.png`	Exp 1	Quad-tree rectangles + all deliveries	Regions' spatial footprint and depth; centroids marked	Helps see clustering and region sizes
`Clean_Path_Route_Size_{n}.png`	Exp 1	Final S→N route on streets; visited vs unvisited points	Title includes total distance and coverage %	Corridor band implicit (not drawn)
`Clean_Path_Scalability_Analysis.png`	Exp 1	4 panels: time vs n, #regions vs n, coverage %, efficiency	Expect time and #regions to grow with n; coverage plateaus reveal corridor tightness	Efficiency = deliveries/km
`experiment_2_results.csv`	Exp 2	Numeric metrics per depot pair	Distance/time/memory/coverage per pair	Useful for tables in reports
`Multi_Depot_Performance_Heatmap.png`	Exp 2	Heatmap (distance, time, memory, covered)	Brighter/larger values per metric per depot pair	Values are normalized by colormap, exact text overlays included
`Delivery_Points_Covered_by_Depot_Pair.png`	Exp 2	Bar chart of covered deliveries per pair	Compare which OD pairs are most effective	Sensitive to corridor width & city geometry
`Regional_Distribution_Analysis.png`	Exp 2	Deliveries by *nearest depot* (bar chart)	Which depot "owns" more nearby points	Heuristic by haversine, not network distance
`Priority_All_Points.png`	Exp 3	All points colored by priority	Red=high, orange=med, green=low	No regions yet; useful sanity check

Artifact / Pattern	Produced by	What it shows	How to read	Notes/Caveats
`Priority_Selected_Regions_{cfg}.png`	Exp 3	Priority-colored points + **only selected** region rectangles	Confirms which regions the selector kept	Selection is by score & size only (no corridor)
`Priority_NoCorridor_Route_{cfg}.png`	Exp 3	Street route for priority-only selection	Coverage tilts toward high-priority clusters	Distance may be larger than Exp 1
`Priority_NoCorridor_Coverage_Comparison.png`	Exp 3	Bar chart of total deliveries covered per config	Compare strict vs moderate selectors	Use with distance to see trade-offs
`priority_no_corridor_results_sn.csv`	Exp 3	Metrics per config (distance, coverage, efficiency)	Join with visuals for narrative	File naming indicates "no corridor" run

---

6) Interpretation guide

- **Coverage (%)**: visited deliveries ÷ total deliveries. Higher is better, but watch the corresponding **distance (km)**.
- **Efficiency (deliveries/km)**: how many deliveries you get per km of street travel. Useful to compare configurations independent of absolute scale.
- **#Regions used**: rough proxy for how fragmented the route becomes. Too many small regions can inflate stitch costs; too few big regions can cause detours.

---

7) Known limitations & future improvement ideas

- **Corridor (Exp 1/2)** may ignore cheap high-value clusters slightly off the spine. Option: allow soft scoring that trades detour for high priority.
- **Centroid distance** can misrepresent elongated regions. Option: min distance from **any** region point or convex hull to spine.
- **Nearest-neighbor tours** are fast but suboptimal. Option: a tiny **2-opt** pass per region (often saves 10–25% intra-region distance).

- **Feasibility/budget** is not enforced in these experiments; add an end-reachability reserve check if needed for production constraints.
- **Dijkstra cache** memory: large LRU size can blow up resident memory on big graphs; trim the cache or cache only distances to delivery/depots.
- **Fallback u→v hop** on exceptions can draw a straight segment; add connectivity checks or retries on the giant component.

---

8) Repro & tuning checklist

- CSV has `priority` with values in {1,2,3}. If missing, the loader defaults to 2 (neutral).
- For **Exp 1**: tune `corridor_width_km` (narrower = cleaner path, potentially less coverage).
- For **Exp 2**: same method as Exp 1; compare OD pairs fairly by keeping delivery set and parameters fixed.
- For **Exp 3**: set `min_points_per_region`, `top_k_regions` to actually bias toward high-priority clusters; no corridor is used.