The Chinese Postman Problem — 2026 Revision

Revised edition of the 1997 final-year project by Miquel Monserrat Pons (supervisor: Ramón Companys Pascual)

Miquel Monserrat Pons · November 2026

Abstract

This report revises a 1997 final-year project on the Chinese Postman Problem: find the cheapest closed route that covers every street of a network at least once. The method is unchanged in outline: pair the odd-degree junctions by shortest paths at minimum total cost, duplicate those paths, and walk the resulting Eulerian circuit. The 1997 branch-and-bound for the pairing step is replaced by Edmonds' blossom algorithm, which is exact and polynomial; shortest paths use Dijkstra, the circuit uses Hierholzer's linear-time algorithm, the directed version is added through minimum-cost flow, and the mixed version, with one-way and two-way streets together and NP-hard, through branch-and-cut and a new heuristic.

The new Python implementation was tested on classic graphs, on 92 benchmark networks from the arc-routing literature (including real road networks in Lancashire and Flanders), and on synthetic towns of up to 40,000 junctions. Every instance but one, a 40,000-junction town that exceeded the memory limit, was solved to proven optimality. With a second backend that runs Kolmogorov's Blossom V on a sparse candidate graph and adds pairs by dual pricing, a network with 12,066 odd vertices solves in 102 seconds, against about 800 odd vertices in the same time in pure Python. On 63 benchmarks where every street needs service, the single-postman tour costs a median 95.1% of the best known multi-vehicle solution, but as little as 47.5% when capacity is scarce: the CPP is a close bound for fleet routing only while vehicles have capacity to spare. For mixed networks, branch-and-cut proves the optimum on every benchmark network within 5.8 seconds on all 20 towns of up to 400 junctions and on 8 of the 10 towns of 900 and 1,600 junctions, and the heuristic averages 0.12% above the optimum on the benchmarks and 1.35% on the towns. The report closes by following the paths left open in 1997: streets whose

first pass costs more than the next, streets that need no service (the rural postman problem) and several postmen sharing a district, each with its complexity, an exact method or a heuristic with a proven bound, and the measured distance from the optimum.

A JavaScript port of the solver runs in the web browser, so anyone can solve their own street network, from a file or straight from OpenStreetMap, at miquelmp.github.io/chinese-postman-problem.

---

1. Introduction

The Chinese Postman Problem (CPP) asks for the cheapest closed walk that traverses every edge of a graph at least once. The 1997 project solved it by duplicating edges until every vertex has even degree, choosing those duplicates with a custom branch-and-bound (B&B) search. This revision keeps the same overall method but replaces the B&B with an exact, polynomial-time matching algorithm, and tests the result on a much wider set of instances. The polynomial algorithms used here were all published before 1997; that project set out to study a branch-and-bound alternative. What has changed since is the engineering: implementations such as Blossom V, compiled libraries and integer-programming solvers reach far larger networks.

The problem is named after the Chinese mathematician Kwan Mei-Ko (Guan Meigu), who posed it in 1960. Edmonds and Johnson showed in 1973 that the undirected version can be solved in polynomial time by reducing it to a minimum-weight perfect matching.

Typical applications are any service that must cover streets rather than visit points:

- mail and parcel delivery on foot;
- snow ploughing and gritting;
- refuse collection and street sweeping;
- park and street-tree maintenance crews;
- inspection of pipelines, power lines and road markings.

What changed since 1997

Aspect	1997 project	2026 revision
Matching step	Binary B&B over edges, lower bound from row minima	Edmonds' blossom algorithm (exact, polynomial)

Aspect	1997 project	2026 revision
Shortest paths	Bellman-type iteration on the full V × V matrix (the 1997 report's "Fulkerson algorithm")	Dijkstra from each odd vertex only
Eulerian circuit	Hierholzer-style cycle splicing with linear scans of adjacency queues	Hierholzer's algorithm with one pointer per vertex, O(m)
Directed graphs	Proposed as future work	Added, by minimum-cost flow (Edmonds and Johnson, 1973)
Language and platform	Turbo C, MS-DOS, 640 KB memory model	Python 3 with NetworkX, any platform
Practical limit	About 20 odd vertices to proven optimality	About 12,000 odd vertices (40,000 junctions) in under two minutes with Blossom V; about 800 in pure Python
Mixed streets (one-way and two-way)	Not covered	Exact branch-and-cut (HiGHS) and a heuristic with local search

Contributions of this revision

1. A clean, tested reimplementation of the full CPP pipeline.
2. A computational study on nine families of instances, none taken from the original Vilanova i la Geltrú case studies.
3. A comparison of the 1997 branch-and-bound with the exact method, and a tightened lower bound for anyone who extends the B&B to harder variants.
4. The paths left open in 1997, implemented and measured: directed and mixed networks, service and travel costs, the rural postman problem, several postmen and capacitated fleets.

---

2. Graph-theoretic background

Only four ideas are needed: graphs with edge costs, vertex degree, Euler's theorem and matchings. The 1997 report spent a chapter on general nomenclature; this revision keeps just what the method uses.

Graph. G = (V, E) has a vertex set V and an edge set E, with n = |V| and m = |E|. In an undirected graph each edge joins two vertices without orientation; in a directed graph each arc

goes from a tail to a head. A multigraph allows several parallel edges between the same pair. Each edge e carries a cost c(e) > 0, such as length, time or fuel.

Degree. The degree of a vertex is the number of edge ends that meet it. Because every edge has two ends, the sum of all degrees is 2m, so every graph has an even number of odd-degree vertices. This handshake lemma is why the odd vertices can always be paired up.

Walks and circuits. A walk is a sequence of vertices and edges, starting and ending at a vertex, in which each edge joins the vertex before it to the vertex after it. A closed walk returns to its start. An Eulerian circuit is a closed walk that uses every edge exactly once.

Euler's theorem. A connected multigraph has an Eulerian circuit if and only if every vertex has even degree. For a strongly connected directed multigraph, the condition is that every vertex has in-degree equal to out-degree. Euler (1741) proved that even degrees are necessary and stated the converse without proof; Hierholzer (1873) published the first proof of it.

Matchings. A matching is a set of edges no two of which share a vertex. It is perfect when it covers every vertex. Given edge costs, a minimum-weight perfect matching is a perfect matching of least total cost. On a complete graph with an even number of vertices a perfect matching always exists.

Shortest paths. d(u, v) is the cost of a cheapest path from u to v. With non-negative costs it is computed by Dijkstra's algorithm (1959) per source in O((n + m) log n) time with a binary heap, or O(m + n log n) with a Fibonacci heap.

---

3. The Chinese Postman Problem

The optimal postman tour costs the sum of all edges plus the cheapest way to pair up the odd vertices by shortest paths. Everything else in this report is about computing that pairing and then walking the resulting graph.

3.1 Formulation

Given a connected undirected multigraph G = (V, E) with costs c(e) > 0, find a closed walk that traverses every edge at least once and has minimum total cost. Equivalently, choose how many

extra copies x(e) ≥ 0 of each edge to add so that the augmented multigraph is Eulerian, at minimum added cost:

$$\min \sum_{e \in E} c(e)\, x(e) \quad \text{s.t.} \quad \deg_G(v) + \sum_{e \ni v} x(e) \equiv 0 \pmod 2 \;\; \forall v \in V, \qquad x(e) \in \mathbb{Z}_{\geq 0}$$

An optimal solution never uses more than one extra copy of an edge, so x(e) ∈ {0, 1}. The set of duplicated edges is a minimum-cost T-join, where T is the set of odd vertices: a set of edges in which exactly the vertices of T have odd degree.

3.2 Edmonds' method

1. Let T be the set of odd-degree vertices. If T is empty, G is already Eulerian; go to step 5.
2. Compute shortest-path distances d(u, v) for every pair u, v in T.
3. Build the complete graph K_T on T with edge costs d(u, v), and find a minimum-weight perfect matching M* on it.
4. For each pair (u, v) in M*, duplicate every edge on a shortest u–v path in G.
5. The augmented multigraph has all degrees even; find an Eulerian circuit in it. That circuit is an optimal postman tour.

The total cost is c(E) + c(M*). The method is exact for the following reason. Any set of duplicated edges that makes all degrees even is a T-join, and every T-join contains edge-disjoint paths that pair up the vertices of T. Each of those paths costs at least the shortest-path distance between its ends, so no T-join costs less than c(M*). The paths chosen in step 4 reach that value: with positive costs they cannot share an edge, because removing both copies of a shared edge would keep every parity and give a T-join cheaper than c(M*).

3.3 Worked example from 1997

The 1997 report used a small park-maintenance network with 8 vertices and 12 edges. Its edge costs, taken from the street plan, are:

Edge	1–8	1–7	8–7	1–4	7–4	7–6	4–5	6–5	6–3	3–5	5–2	3–2
Cost	2	1	5	6	4	7	2	1	4	7	6	9

As in the original report, costs stand for travel time or fuel rather than plain distance.

The odd vertices are T = {1, 3, 4, 6}. The three possible perfect matchings on K_T cost:

Matching	Cost
(1,3) + (4,6)	12 + 3 = 15
(1,4) + (3,6)	5 + 4 = 9
(1,6) + (3,4)	8 + 7 = 15

The optimum is M* = {(1,4), (3,6)}. It duplicates the path 1–7–4 and the edge 3–6. The original network costs 54, so the optimal tour costs 54 + 9 = 63 and has 15 edges. The 2026 code reproduces this exactly and returns the tour 1–8–7–1–7–4–7–6–3–2–5–3–6–5–4–1.

---

4. Minimum-weight perfect matching

The matching step is solved exactly, in polynomial time, by Edmonds' blossom algorithm. This replaces the 1997 branch-and-bound entirely: there is no longer any instance size at which the method has to fall back on a time limit or a heuristic.

4.1 Linear programming formulation

Let K = (T, F) be the complete graph on the odd vertices, with cost d(e) on each edge, and let x(e) = 1 when edge e is in the matching. For any vertex set S, δ(S) is the set of edges with exactly one end in S. Edmonds (1965) showed that the following linear program has an integral optimum, so it describes perfect matchings exactly:

$$\min \sum_{e \in F} d(e)\, x(e) \quad \text{s.t.} \quad x(\delta(v)) = 1 \;\; \forall v \in T, \qquad x(\delta(S)) \geq 1 \;\; \forall S \subseteq T,\ |S| \geq 3 \text{ odd}, \qquad x \geq 0$$

Without the odd-set constraints the program can return half-integral solutions on odd cycles: three vertices joined in a triangle, each edge at 0.5. The odd-set constraints rule these out. There are exponentially many of them, but the algorithm never lists them; it only touches the few that matter.

4.2 How the blossom algorithm works

The algorithm is a primal-dual method. It keeps a matching, which starts empty, and a feasible dual solution: a price y(v) on each vertex and a price z(B) on each odd set B in use. An edge is

tight when the prices of its ends, plus those of the odd sets it crosses, equal its cost. The algorithm repeats three moves until every vertex is matched:

1. **Grow.** From an unmatched vertex, build an alternating tree using tight edges only. If it reaches another unmatched vertex, flip the path between them: the matching grows by one edge.
2. **Shrink.** If the tree closes an odd cycle (a blossom), contract the cycle into a single pseudo-vertex and keep growing.
3. **Adjust prices.** If the tree is stuck, change the dual prices by the largest step that keeps them feasible. This makes at least one new edge tight, or lets a pseudo-vertex be expanded again.

When every vertex is matched, the matching and the prices satisfy complementary slackness, which proves optimality. Chapter 3 of the 1997 report already presented this algorithm. The 2026 version uses it directly, through a mature library implementation, so the matching no longer needs a purpose-built search.

4.3 Complexity and implementations

Implementation	Language	Worst-case time	Notes
Edmonds (1965)	—	$O(n^4)$	Original algorithm
Gabow (1973), Lawler (1976)	—	$O(n^3)$	Standard textbook bound
Galil, Micali and Gabow (1986)	—	$O(nm \log n)$	Better on sparse graphs
Blossom V, Kolmogorov (2009)	C++	Believed $O(n^3m)$, no better than Blossom IV; much faster in practice	Fast backend in this revision; non-commercial licence
NetworkX `min_weight_matching`	Python	$O(n^3)$	Default backend in this revision

Here n is the number of odd vertices and m the number of edges of K, so $m = n(n - 1)/2$. NetworkX was chosen for readability and portability: the undirected and directed solvers take 133 lines of Python, and the Eulerian circuit 42 more. Its pure-Python matching becomes the bottleneck above a few hundred odd vertices (section 7.4), so large networks use a second backend built on Blossom V, described next.

4.4 Large networks: a sparse candidate graph with dual pricing

The complete graph K has about $n^2/2$ edges, yet an optimal matching almost always pairs each odd vertex with one of its nearest odd neighbours. The fast backend exploits this without giving up the optimality guarantee:

1. Compute shortest-path distances from every odd vertex with SciPy's compiled Dijkstra.
2. Build a sparse candidate graph: each odd vertex's 10 nearest odd vertices, plus the pairs of a greedy matching so that a perfect matching is sure to exist.
3. Solve it with Blossom V, Kolmogorov's C++ implementation, and read the optimal dual prices y.
4. Pricing: scan all pairs (u, v) not in the graph and add those with negative reduced cost, d(u, v) < y(u) + y(v), counting blossom prices. Re-solve and repeat.
5. When no pair has negative reduced cost, the duals are feasible for the whole complete graph, so the matching is optimal for it too.

This is column generation applied to matching, the same strategy Blossom V uses for its geometric instances. On the test networks the first candidate graph comes very close: at most 26 pairs had to be added, in up to two pricing rounds, followed by a last round that confirms optimality. Related ideas reach even further: the sparse blossom algorithm of Higgott and Gidney (2025), developed for quantum error correction, matches millions of vertices without ever building distances between all pairs.

Blossom V may be used for evaluation and non-commercial research only and may not be redistributed. The package therefore ships only a small C wrapper and a script that downloads and compiles it; NetworkX stays the default backend, and commercial use would need a licence from UCL Business or a free alternative such as LEMON (Boost licence).

4.5 Milestones in matching, 1965–2026

Minimum-weight perfect matching has kept improving since Edmonds' blossom algorithm, both in theory and in working code. The table lists the milestones most relevant to postman problems, dated by publication.

Year	Authors	Contribution	Relevance here
1965	Edmonds	Blossom algorithm and the matching polytope	Foundation of the exact method, $O(n^4)$

Year	Authors	Contribution	Relevance here
1973–1976	Gabow; Lawler	$O(n^3)$ implementations	Textbook bound, also that of NetworkX
1980	Micali and Vazirani	$O(m\sqrt{n})$ maximum-cardinality matching	Fastest bound for the unweighted case
1986	Galil, Micali and Gabow	$O(nm \log n)$ weighted matching	Faster on sparse graphs
1990	Gabow	$O(nm + n^2 \log n)$	Still the best strongly polynomial bound
1991	Gabow and Tarjan	Cost scaling for integer weights	Near $O(m\sqrt{n} \log(nN))$
1999	Cook and Rohe	Blossom IV, with variable ε and pricing	Solved geometric instances of up to 5 million points
2002	Mehlhorn and Schäfer	$O(nm \log n)$ implementation in LEDA	Data structures that make the bound practical
2009	Kolmogorov	Blossom V	Fast backend of this revision
2011	Dezső, Jüttner and Kovács	LEMON graph library	Free alternative with a permissive licence
2014	Duan and Pettie	$(1 - \varepsilon)$-approximate weighted matching in linear time	For maximum-weight matching; gives no guarantee for a minimum-cost perfect pairing
2018	Duan, Pettie and Su	Exact scaling algorithm, $O(m\sqrt{n} \log(nN))$	Best known bound for integer costs
2022	Chen, Kyng, Liu, Peng, Probst Gutenberg and Sachdeva	Minimum-cost flow in almost-linear time	Directed CPP in almost-linear time, in theory
2023	Wu and Zhong	Fusion Blossom, parallel matching	Quantum error correction at scale
2025	Higgott and Gidney	Sparse Blossom	Millions of vertices without all-pair distances
2026	Arkhipov and Kolmogorov	Blossom VI	Reported faster than Blossom V on the hardest instances tested; not tried here

Here N is the largest integer cost. Two lessons follow for the CPP. First, the worst-case bounds have improved far more than road networks need: with the sparse pricing of section 4.4, a well-engineered code is limited by the memory for distances, not by the matching. Second, the most active work now comes from quantum error correction, where decoders must pair millions of

defects per second. Their local, sparse search is the natural next step for very large postman instances.

---

5. Eulerian circuits

Once every degree is even, Hierholzer's algorithm (1873) writes out the tour in time proportional to the number of edges. It is the algorithm the 1997 project already sketched in its section 5.2.2, now implemented so that each edge is looked at once.

The idea follows the constructive proof of Euler's theorem. Walk from the start vertex along unused edges until you are stuck; with all degrees even, you can only get stuck back at the start, so you have a closed sub-tour. Then return along that sub-tour to the first vertex that still has unused edges, walk a new sub-tour from there and splice it in. Repeat until no unused edges remain.

The iterative version uses a stack:

1. Push the start vertex.
2. Look at the vertex v on top of the stack. Advance v's pointer past any edges already used.
3. If v has an unused edge (v, w), mark it used and push w.
4. Otherwise pop v and append it to the output.
5. When the stack is empty, the output reversed is the Eulerian circuit.

Each vertex keeps a pointer into its adjacency list that only moves forward, and each edge is marked used once. The total work is therefore O(n + m). This per-vertex pointer is the refinement that makes the 2026 implementation strictly linear in the size of the graph.

Fleury's rule, discussed in the 1997 report, also works but must test at each step whether an edge is a bridge of the remaining graph. Even with modern dynamic connectivity structures this is slower and more complicated than Hierholzer, so it is no longer used in practice.

6. Implementation

The solver is a small Python 3 package, `chinese_postman`, built on NetworkX and covered by 273 automated tests (218 when the optional Blossom V library is not built). It replaces the six Turbo C modules of 1997 (about 1,000 lines plus custom tree, list and matrix structures) with eleven modules of plain Python and an optional compiled matching library. A JavaScript port of the same solver runs the interactive website (section 6.6).

6.1 Modules

Module	Replaces (1997)	Responsibility
`io.py`	`GRAFO.C`, `leer_archivo`	Reads and writes the original `.GRA` format; reads the CARP `.dat` benchmark format
`cpp.py`	`MAIN.C`, `FULKER.C`, `ARBOL.C`, `SIMETR.C`	Undirected CPP (Edmonds' method) and directed CPP (minimum-cost flow)
`fast.py`	— (new)	Large networks: SciPy Dijkstra and Blossom V with dual pricing, called through a small C wrapper in native/
`mixed.py`	— (new)	Mixed CPP: bounds, exact branch-and-cut (HiGHS via SciPy), heuristic with local search, mixed instance generators
`bnb.py`	`ARBOL.C`, `MAIN.C`	The branch-and-bound of section 8.3, kept to measure the search; the solvers do not use it
`euler.py`	`EULERIAN.C`	Hierholzer's algorithm on multigraphs, directed or not
`instances.py`	— (new)	Generators for the test families of chapter 7
`web.py`	— (new)	Builds networks for the website from examples, .GRA and CARP files and OpenStreetMap; reference for the JavaScript port
`rural.py`	— (new)	Service and travel costs; rural postman: Frederickson's heuristic, lower bounds, branch-and-cut, combined solve_rural (chapter 10)
`fleet.py`	— (new)	Several postmen (optimal split, local search, bounds, exact model for small networks) and capacitated fleets
`sequence.py`	— (new)	Routes as sequences of served streets: cost, best directions, insertion, 2-opt

Graphs are NetworkX multigraphs, so parallel streets between the same two junctions are kept, as the 1997 adjacency lists allowed. When computing shortest paths only the cheapest of any parallel edges matters, and the solver works on that simple graph internally.

6.2 Pipeline

1. Check that the graph is connected and collect the set T of odd-degree vertices.
2. Run Dijkstra from each vertex of T, keeping distances and paths to the other vertices of T: O(|T| (n + m) log n) with NetworkX's binary heap.
3. Build the complete graph on T and call `min_weight_matching`.
4. Add one copy of every edge on the shortest path of each matched pair.
5. Run Hierholzer on the augmented multigraph and return the tour.

The result object holds the original, added and total costs, the duplicated edges, the matched pairs, the tour as a sequence of vertices, and the time spent in each phase.

6.3 Directed graphs

For a directed graph the imbalance of each vertex, out-degree minus in-degree, plays the role of odd degree. Vertices with more arcs in than out must send extra walks to vertices with more arcs out than in. Choosing those walks at least cost is a transportation problem, solved with NetworkX's `min_cost_flow` on the street network itself, with unbounded arc capacities. The flow on each arc is the number of extra copies to add. The graph must be strongly connected. This reduction goes back to Edmonds and Johnson (1973).

6.4 Testing

Test group	Cases	What it checks
1997 example	1	Added cost 9, total 63, matching {(1,4), (3,6)}
Classic graphs	3	Königsberg, Petersen and dodecahedron give the known optimum
Brute force	30	Random graphs with up to 8 odd vertices: same cost as enumerating every pairing
Benchmark bound	23	On gdb, where every edge is required, the CPP never exceeds the published CARP lower bound
Structure	4	Parallel edges, generated cities and directed grids give valid closed tours
Blossom V backend	55	Same optimum as NetworkX on 51 benchmark files, and on towns where a one-neighbour start forces several pricing rounds
Mixed CPP	41	Exact equals brute-force enumeration on 25 random networks; bound ≤ optimum ≤ heuristic; all-two-way and all-one-way cases equal the pure CPPs; the combined strategy proves optimality, including through the cutoff
Branch-and-bound	36	Bounds of the 1997 worked example; same optimum as blossom on 30 random cases (10 for each of the three variants); strict node limit

Test group	Cases	What it checks
Web builders	13	Examples, .GRA, CARP and OpenStreetMap input give the expected networks and solve; bad input gives a clear message
JavaScript solver	243 networks	The website solver gives the Python optimum on every undirected and directed case, never claims a false optimum on mixed ones, and returns a valid tour
Rural postman	36	Exact equals enumeration of every multiset of extra passes on the original network; Frederickson within 3/2 of the optimum; every street served; all-required equals the CPP; first-pass costs, undirected and directed
Several postmen	31	Bound ≤ exact optimum ≤ heuristic; split within 2 − 1/k of the optimum; split equals enumeration of cut points; one postman equals the rural postman; capacities respected
JavaScript extensions	91 networks	Website rural postman and several postmen: valid routes, every street served, never below the Python optimum, bounds never above it

Every test that produces a tour also verifies it: it is closed, it covers every original edge, and its length equals the original edges plus the duplicates.

6.5 Usage

```
from chinese_postman import read_gra, solve_undirected

sol = solve_undirected(read_gra("examples/example_1997.gra"), start=1)
print(sol.summary())
```

6.6 Web version

The project website, miquelmp.github.io/chinese-postman-problem, lets anyone solve a postman problem in the browser: one of the built-in examples or benchmark networks, a `.GRA` file, or a district chosen on an OpenStreetMap map. For an immediate answer the solver is ported to JavaScript and runs inside the page, with no server and nothing to install.

Part	File	Content
Matching	`site/assets/cpp.js`	Edmonds' blossom algorithm, exact, following Van Rantwijk's reference implementation
Directed and mixed streets	`site/assets/cpp.js`	Minimum-cost flow; lower bounds and the mixed heuristic of chapter 9, with its starting orientations, balancing and local search
Tour	`site/assets/cpp.js`	Hierholzer's algorithm, as in chapter 5

Part	File	Content
Networks	`site/assets/networks.js`	Built-in examples, `.GRA` and CARP readers, OpenStreetMap import through the Overpass API
Extensions	`site/assets/extensions.js`	Rural postman and several postmen (chapter 10): Frederickson's heuristic, local search, optimal split into routes, lower bounds

The port is checked against the Python package. On 243 networks, from classic graphs to benchmark files and mixed towns, it returns the same optimum in every undirected and directed case, stays a mean 0.25% above the optimum on mixed ones, and always returns a valid closed tour; on 3,000 random matchings it agreed with NetworkX every time. The page is ready to solve in under a second.

Only one step still needs Python: proving that a mixed tour is optimal. On request the page loads Pyodide, Python compiled to WebAssembly, with SciPy and the `chinese_postman` package, and runs the branch-and-cut of chapter 9 inside the browser in a few seconds.

The interactive version accepts files of up to 2,500 junctions and OpenStreetMap areas of up to about one square kilometre, or 1,500 junctions. Larger networks are better solved with the Python package and its Blossom V backend. The page's service settings choose which streets need service (every street, a random share, the main roads of an OpenStreetMap area, or those marked in a benchmark file) and how many postmen share them, with the methods of chapter 10.

7. Computational experiments

Every instance tested, 92 distinct benchmark networks (53 of them from real road data) and synthetic cities with up to 40,000 junctions, was solved to proven optimality, apart from one 40,000-junction town that exceeded the memory limit (section 7.4). None comes from the 1997 Vilanova i la Geltrú case studies. All runs used one core of a 2026 cloud Linux container, Python 3 and NetworkX 3.7; `results/environment.json` records the machine of the latest run (Intel Xeon at 2.10 GHz, 2 cores, 7.8 GB). The timings come from separate runs, so they are comparable within a table and not across chapters.

7.1 Instance families

Family	Source	Networks	Junctions	Streets	Odd vertices
Classic graphs	Königsberg, Petersen, dodecahedron, 1997 example	4	4–20	7–30	4–20
gdb	Golden, DeArmon and Baker (1983), random networks	23	7–27	11–55	0–16
val	Benavent et al. (1992), generated networks	10	24–50	34–97	12–28
kshs	Kiuchi et al. (1995)	6	6–10	15	4–6
egl	Li and Eglese (1996), Brandão and Eglese (2008): winter-gritting roads in Lancashire, UK	3	77–255	98–375	50–190
Beullens C, E (D, F share their networks)	Beullens et al. (2003): road network of Flanders, Belgium	50	26–97	35–142	14–54
Grid city	Square grid, short blocks and long avenues, 10% of segments closed	12	100–40,000	161–71,752	42–12,066
Delaunay town	Random junctions, Delaunay triangulation thinned to 75%	11	100–20,000	214–44,974	48–10,006
One-way grid	Directed grid, alternating one-way streets, two-way outer ring	5	48–3,600	113–8,142 arcs	18–204 unbalanced

Benchmark files that share a road network and differ only in demands (for example val1A, val1B and val1C) were solved once, leaving 92 distinct networks out of 254 files. Families A and B in the same repository, of unknown origin, were left out. Delaunay towns are drawn in a 1 km square whatever their size, so the largest have unrealistically short streets, about 6 m on average at 20,000 junctions. Lengths are rounded to whole metres, which gives 152 of those 44,974 streets a cost of 0.

7.2 Classic graphs

Graph	Odd vertices	Streets	Added	Total	Tour length
1997 example	4	12	9	63	15 edges
Königsberg bridges	4	7	2	9	9 crossings
Petersen graph	10	15	5	20	20 edges
Dodecahedron	20	30	10	40	40 edges

Euler's citizens of Königsberg could not cross each bridge exactly once, but a postman can cover all seven with nine crossings by repeating two. In the Petersen graph and the dodecahedron every vertex is odd, and both contain a perfect matching of unit edges, so the tour repeats one edge for every two vertices: a third of the edges.

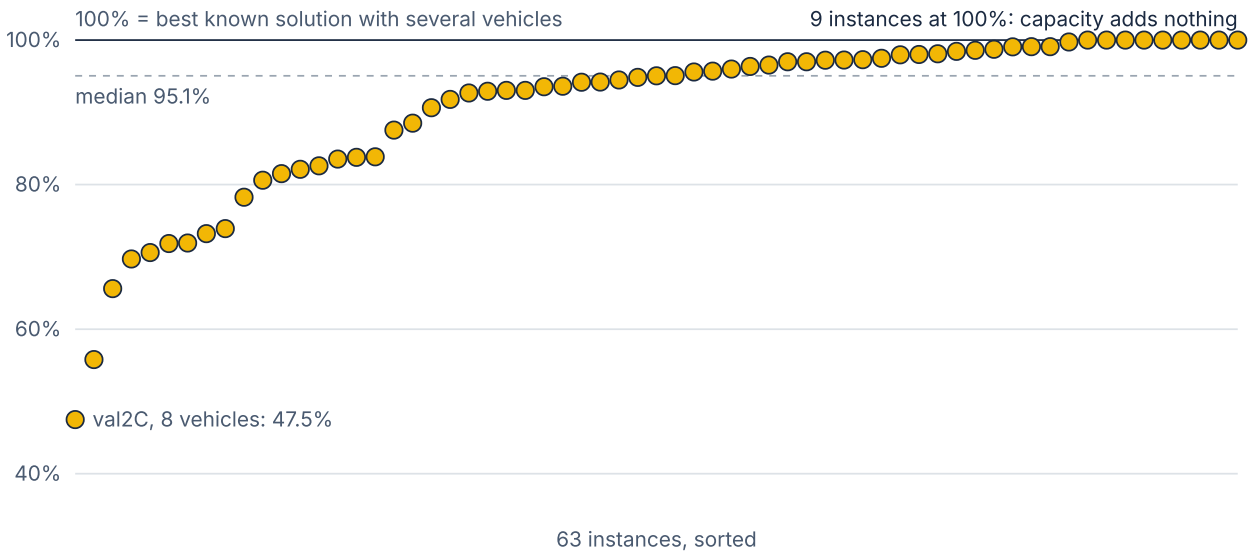
7.3 Benchmark networks

Family	Networks	Added distance, % of network (mean)	Range	Slowest solve
gdb	23	8.9%	0–22.2%	0.003 s
val	10	14.9%	10.9–18.5%	0.007 s
kshs	6	14.9%	10.5–20.4%	0.001 s
Beullens C	25	20.1%	11.4–24.3%	0.06 s
Beullens E	25	19.4%	12.3–26.9%	0.06 s
egl-e	1	37.4%	—	0.04 s
egl-s	1	24.5%	—	0.21 s
egl-g	1	24.4%	—	2.46 s

On real road networks a postman must repeat about a fifth of the distance; on the sparse Lancashire network of egl-e, more than a third. The largest benchmark, egl-g, with 255 junctions and 190 odd vertices, takes 2.5 seconds, with optimality proven.

In 39 of these networks every street needs service, so any multi-vehicle CARP solution is also a postman tour, and the CPP optimum is a lower bound for it. Files that share a road network were solved once above, but they differ in capacity and number of vehicles, so the comparison with fleets uses all 63 gdb, val and kshs files in which every street is required. The solver respects the bound on all 63. How close it comes depends on capacity:

One postman costs a median 95.1% of the best known fleet, and as little as 47.5%



results/cpp_bound.csv · 63 all-required instances from gdb, val and kshs; best known CARP solutions as published in CARPData.jl

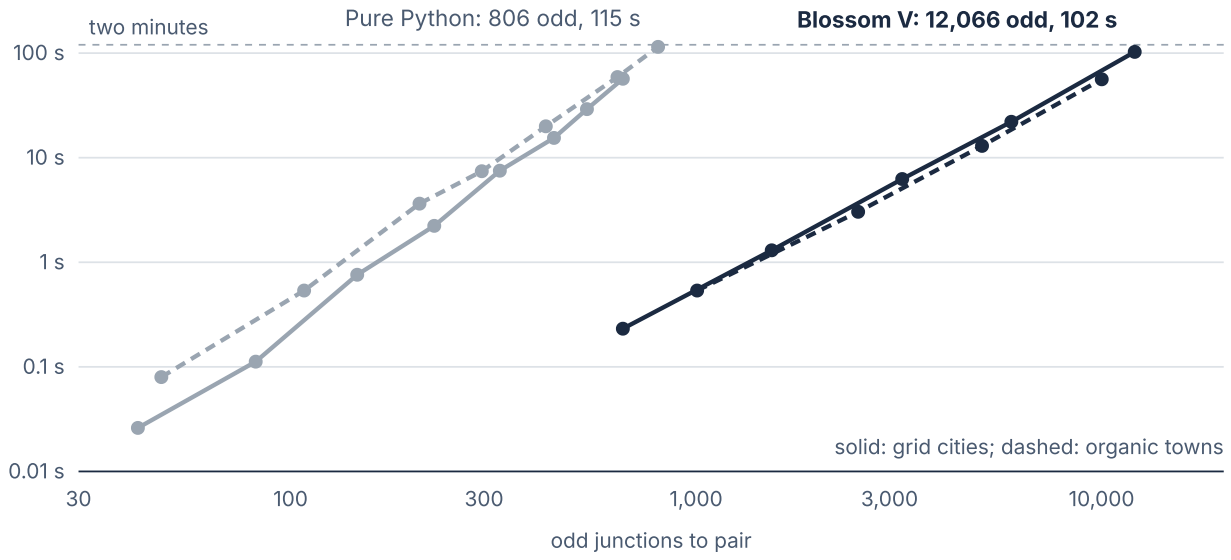
Instances	Number	Median	Lowest
gdb	23	96.0%	71.8%
kshs	6	88.3%	65.6%
val A (fewest vehicles)	10	99.1%	95.1%
val B	10	96.0%	83.8%
val C	10	78.7%	47.5%
val D (four networks only)	4	76.9%	71.9%
All	63	95.1%	47.5%

The mean is 90.3%; 19 instances fall below 90% and 10 below 80%, while 9 reach 100%, where capacity adds nothing to the postman's tour. On each of the 10 val networks the bound weakens steadily as capacity shrinks and vehicles are added: val2 goes from 95.6% with 2 vehicles to 47.5% with 8, because every vehicle must drive out from and back to the depot. Vehicle count does not explain everything: the lowest kshs and gdb values are kshs1 (65.6%, 4 vehicles) and gdb8 (71.8%, 10 vehicles), while gdb23 stays at 95.7% with 10 vehicles. In all 63 files the published lower and upper bounds coincide, so these fleet costs are proven optima.

7.4 Scaling

With the Blossom V backend, a network with 40,000 junctions and 12,066 odd vertices solves to proven optimality in 102 seconds, against about 800 odd vertices in the same time with NetworkX. These two figures come from different families; within one family, Blossom V reaches 12,066 odd vertices in 102 s on grids, against 660 in 57 s with NetworkX, and 10,006 in 56 s on Delaunay towns, against 806 in 115 s. Grid cities and Delaunay towns were generated with up to 40,000 and 20,000 junctions respectively; both backends were run, NetworkX until a case passed 90 seconds.

With Blossom V, two minutes now covers 12,000 odd junctions instead of 800



results/scaling.csv (NetworkX, 15 networks) and results/scaling_blossom5.csv (Blossom V, 9 networks) · 100–40,000 junctions

Network	Junctions	Odd vertices	Shortest paths	Matching	Tour	Total	Pricing rounds	Peak memory
Grid city	2,025	660	0.1 s	0.01 s	0.1 s	0.2 s	1	0.1 GB
Grid city	10,000	3,224	3.9 s	0.2 s	2.1 s	6.2 s	2	0.4 GB
Delaunay town	10,000	5,072	8.0 s	0.8 s	4.1 s	13 s	3	not measured separately
Grid city	19,600	5,988	14.4 s	0.8 s	6.8 s	22 s	2	1.1 GB
Delaunay town	20,000	10,006	35.4 s	2.7 s	18.0 s	56 s	2	not measured separately

Network	Junctions	Odd vertices	Shortest paths	Matching	Tour	Total	Pricing rounds	Peak memory
Grid city	40,000	12,066	64.7 s	4.7 s	33.1 s	102 s	3	3.4 GB

On the 660-odd-vertex grid the fast backend is 245 times faster than NetworkX, and the matching itself takes 0.01 s instead of 52 s. Starting from 10 nearest neighbours, one to three rounds were always enough, the last of them only confirming optimality; seven of the nine networks needed extra pairs, at most 26. The matching is no longer the bottleneck: shortest paths take about 60% of the time and building the tour about 30%, both still written in Python around compiled kernels and both easy to speed up further. Memory is dominated by the table of distances between odd vertices, which grows with their square, and by two temporary matrices of the same shape used to pick the nearest neighbours; at 7 GB of RAM the ceiling lies between the 12,066 odd vertices solved and the 19,942 that failed. A Delaunay town of 40,000 junctions (19,942 odd vertices) ran out of memory at 7 GB while building the candidate graph, once all its distances had been computed. Solving bigger networks would mean never storing that table, as the sparse blossom method does.

For the NetworkX backend, time grows roughly with the cube of the number of odd vertices, as expected from the $O(n^3)$ matching, which takes over 85% of the time above 200 odd vertices.

The repeated distance settles at 12–15% of the network on both families, lower than on the benchmark networks because grids and triangulations are denser.

7.5 Directed networks

Grid	Junctions	Arcs	Unbalanced junctions	Added distance	Solve time
6 × 8	48	113	18	11.7%	0.003 s
10 × 12	120	280	32	10.4%	0.006 s
20 × 20	400	912	64	8.4%	0.02 s
40 × 40	1,600	3,627	134	6.9%	0.14 s
60 × 60	3,600	8,142	204	6.6%	0.39 s

The directed CPP is much faster than the undirected one at the same size because minimum-cost flow works on the sparse street network itself rather than on a complete graph of odd vertices. In

a city of alternating one-way streets, a postman repeats only 7–12% of the distance.

---

8. From branch-and-bound to blossom: improvements in this version

The 2026 version keeps the structure of the 1997 method and improves each of its stages. The largest gain comes from solving the matching exactly in polynomial time; a tightened lower bound also makes the branch-and-bound itself much faster, which matters for the harder variants of chapter 9 where B&B remains the method of choice.

8.1 Summary of improvements

Stage	1997 version	2026 version	Effect
Matching	Binary branch-and-bound with a time limit for large cases	Edmonds' blossom algorithm	Proven optimum at every size, polynomial time
B&B lower bound	Row minima divided by half the number of odd vertices	Row minima divided by 2	Far fewer nodes; useful for B&B on harder variants
Shortest paths	Bellman-type iteration on the full V × V matrix	Dijkstra from odd vertices only	Work proportional to the odd vertices, not to all pairs
Eulerian circuit	Cycle splicing over adjacency queues	Hierholzer with one pointer per vertex	Linear time
Directed streets	Future work	Minimum-cost flow	Polynomial (Edmonds and Johnson, 1973)
Platform	Turbo C, 16-bit integers, 640 KB memory model	Python 3, unbounded integers, automatic memory	No size limits on costs or data
Verification	Manual checks	273 automated tests in Python and 334 networks and 3,000 matchings re-checked in JavaScript, every tour validated	Results checked on each run

8.2 The 1997 branch-and-bound

Each node of the search tree fixed some pairs of odd vertices as taken and others as rejected. Branching took the cheapest pair still available and created two children: take it or reject it. The lower bound of a node was $c_1 + c_2$, where c_2 is the cost of the pairs already taken and c_1

estimates the cost of completing the matching from the cheapest available pair at each remaining vertex. Nodes were explored lowest bound first while the tree was small, and deepest first afterwards, which gives good solutions early and makes the method a natural time-limited heuristic.

8.3 A tightened lower bound

Every matched pair is reached from both of its ends, so the cost of any perfect matching M is half the sum, over all odd vertices, of the pair each one belongs to. That gives the bound

$$c(M) = \tfrac{1}{2}\sum_{v \in T} d\big(v, M(v)\big) \;\geq\; \tfrac{1}{2}\sum_{v \in T} \min_{u \neq v} d(u, v)$$

The 1997 version divided the same sum by half the number of odd vertices, imp/2. The two coincide when there are four odd vertices, as in the worked example; with more odd vertices the divisor 2 gives a higher, tighter bound and prunes much more of the tree.

Both bounds were measured with a reimplementation of the 1997 search, `chinese_postman/bnb.py`, which follows the Turbo C listings and reproduces the bounds of the 1997 worked example. Only the divisor changes between the two runs. The instances are five grid towns for each number of odd vertices, and the limit is 200,000 explored nodes, so the counts do not depend on the machine:

Odd vertices	1997 bound: proven	Nodes (median)	Tightened bound: proven	Nodes (median)	Nodes (worst)
8	5 of 5	41	5 of 5	6	9
12	5 of 5	1,925	5 of 5	18	55
16	4 of 5	68,058	5 of 5	54	349
20	0 of 5	limit reached	5 of 5	53	1,569
24	0 of 5	limit reached	5 of 5	141	1,052
30	0 of 5	limit reached	5 of 5	1,656	44,740

Even when stopped at the limit, the 1997 search had usually found the optimum: in 14 of its 16 stopped runs, and within 7% in the other two. The counts vary widely between towns of the same size, so they show orders of magnitude and not a precise law.

Two further changes extend the reach: branching on the most constrained vertex, and starting from a greedy pairing improved by pair swaps. With both, the search proves all five towns at 40 and at 50 odd vertices, in under 10 seconds each, and four of five at 60; the tightened bound alone proves four of five at 40 and three of five at 50. They do not help everywhere: at 30 odd vertices the worst case rises from 44,740 to 106,867 nodes. Blossom solves the same cases of up to 50 odd vertices in under 0.1 s, which is why the 2026 version uses it for the CPP and keeps the B&B for problems with no polynomial algorithm.

8.4 Why the B&B still matters

The 1997 conclusion, that a time-limited B&B yields very good solutions quickly on large instances, still holds and carries over to NP-hard arc routing problems such as the mixed CPP and the rural postman problem. For those, the improvements above (a tight bound, a good first solution and a branching rule that fixes the hardest decision first) are exactly the ingredients modern exact methods rely on.

---

9. The mixed Chinese Postman Problem

Real towns mix one-way and two-way streets. This mixed CPP is NP-hard (Papadimitriou, 1976), so no polynomial algorithm is known; the 2026 version solves it exactly by branch-and-cut and approximately by a fast heuristic whose distance from the optimum is measured.

9.1 Why it is harder

The two pure cases are easy for different reasons. An undirected network needs only even degrees, fixed by a matching (chapter 3); a directed one needs only as many arcs in as out at each junction, fixed by a minimum-cost flow (chapter 6). In a mixed network the direction in which each two-way street is driven and the choice of streets to repeat must be decided together, because each decision changes the other's constraints.

A mixed network has a postman tour only if it is strongly connected when two-way streets count in both directions. The generators below make sure of it.

9.2 Model and exact method

Let $x(u \to v)$ be the number of times the tour drives a street from u to v:

$$\min \sum c\,x \quad \text{s.t.} \quad \sum_{\text{out of } v} x = \sum_{\text{into } v} x \;\; \forall v, \qquad x(u \to v) \geq 1 \text{ for a one-way street}, \qquad x(u \to v) + x(v \to u) \geq$$

Any integer solution is a balanced, connected multigraph, so it contains an Eulerian circuit: the tour. The integer program is solved with HiGHS, an open-source solver bundled with SciPy, with its relative gap tolerance set to zero (the default of 0.01% would stop short of a proof). HiGHS runs branch-and-bound with cutting planes: the 1997 idea of separating and bounding, with the bounds supplied by linear programming.

9.3 Bounds and heuristic

Two lower bounds are cheap to compute. The linear relaxation of the model ignores parity and lets a two-way street be driven half each way; the undirected CPP of the network ignores directions. Each misses what the other captures, so the larger of the two is used.

The heuristic is polynomial and runs in four steps:

1. Build four starting orientations of the two-way streets: two roundings of the linear relaxation, and the direction in which an Eulerian circuit of the undirected CPP solution drives each street, read forwards and backwards.
2. For each orientation, add the cheapest extra traversals that balance every junction. With directions fixed this is a minimum-cost flow, solved exactly.
3. Improve the two best by local search. The flow's dual prices m estimate what reversing a street a → b would change, 2·(m_b − m_a); the most promising reversals are tried, each verified by an exact re-solve, and kept only if they lower the cost.
4. Return the best tour found, with the best lower bound as a certificate of how far it can be from the optimum.

The orientation-then-balance idea goes back to Frederickson (1979). The local search is what makes it accurate: across the 121 instances of section 9.4 with a proven optimum, it improved the best starting orientation on 77 and cut the mean gap to the optimum from 1.66% to 0.41% and the worst from 11.9% to 4.8%. Most of the gain is on the benchmarks (mean 1.58% to 0.12%); on the 29 towns the mean falls only from 1.92% to 1.35%.

9.4 Results

On the benchmark networks and on towns of up to 400 junctions the mixed CPP is solved exactly in seconds. Above that, size alone does not decide: some towns are proved in under a

minute and others of the same size are not proved in five. The heuristic is 1% to 3.6% above the best tour of branch-and-cut on the towns left unproved, in a fraction of its time; its worst gap to a proven optimum is 4.8%.

Benchmark networks. In each of the 92 distinct networks of chapter 7, each street was made one-way in a random direction with probability 0.4. Streets that broke strong connectivity went back to two-way, leaving 31% of all streets one-way (24–44% per network).

Result	Value
Networks solved to proven optimality	92 of 92
Slowest exact solve	5.8 s (egl-g, 255 junctions, 375 streets)
Heuristic finds the optimum	78 of 92
Heuristic gap to optimum	mean 0.12%, maximum 2.8%
Slowest heuristic run	3.8 s

Mixed towns. Grid towns in which each whole street is one-way with probability 0.6, as in a typical town centre, from 64 to 6,400 junctions; five towns per size, a 5-minute limit for the exact method.

Junctions	Streets	Proved in 5 min	Exact: time when proved	Heuristic: time	Heuristic: gap to the optimum, when proved	Unproved: exact to its bound	Unproved: heuristic above the exact tour
64	112	5 of 5	under 2 s	0.1–0.4 s	0–2.6%	—	—
144	264	5 of 5	4.1–5.2 s	0.3–1.1 s	0.2–2.8%	—	—
256	480	5 of 5	2.4–7.5 s	0.4–1.0 s	0.3–2.8%	—	—
400	760	5 of 5	1.6–5.5 s	0.6–2.8 s	0.2–1.8%	—	—
900	1,740	4 of 5	9–190 s	3.4–8.4 s	0.7–3.3%	0.47%	3.6%
1,600	3,120	4 of 5	16–97 s	7–22 s	1.6–4.8%	0.14%	2.0%
3,600	7,080	1 of 5	38 s	25–51 s	2.3%	0.48–1.01%	1.0–2.4%
6,400	12,640	0 of 5	—	64–98 s	—	0.25–0.97%	1.0–2.0%

Three conclusions follow.

1. **Exact is practical up to a district, but not always.** Branch-and-cut proves every benchmark network and all 20 towns of up to 400 junctions within 8 seconds, 8 of the 10

towns of 900 and 1,600 junctions, one of five at 3,600 and none at 6,400. Every unproved town still ends within about 1% of its proven bound in five minutes, with a certificate.

2. **The heuristic is fast, and more accurate on the benchmarks than on towns.** With a proven optimum it averages 0.12% above it on the 92 benchmarks (worst 2.8%) and 1.35% on the 29 towns (worst 4.8%). Its worst cases are towns where single-street reversals get stuck; moves that reverse whole cycles of two-way streets would be the next improvement.
3. **Bounds matter.** The cheap combined bound averages 94.8% of the optimum on the benchmarks, as low as 83%, so on its own it certifies little. The bound computed by branch-and-cut is far tighter: it is what turns a good solution into a certified one.

9.5 Combined strategy

In practice the two methods are used together, through a single function, `solve_mixed`, that returns the best tour found and a proven lower bound:

1. Compute the cheap bounds and run the heuristic, which gives a good tour in seconds.
2. If the heuristic already matches the bound, or is within a chosen tolerance of it, stop.
3. Otherwise run branch-and-cut for the remaining time, asking it only for tours cheaper than the heuristic's. If none exists the model is infeasible, which proves the heuristic tour optimal; if one exists, it replaces the heuristic tour.

The objective cutoff lets branch-and-cut discard every branch that cannot beat the heuristic, so the heuristic's quality speeds up the exact search. On a 64-junction town where the heuristic was already optimal, this proves it in under a second. On a 1,600-junction town where the heuristic alone was 4.8% above the optimum, the combination returns the optimum in 79 seconds. With a tolerance of 0.5%, towns of up to 256 junctions stop in about three seconds or less with a tour certified within that margin.

This extends the approach of chapter 12 of the 1997 report, a time-limited exact search used as a heuristic: there is now a branch-and-cut solver behind it, and a proven bound on the remaining error.

10. The paths left open in 1997

The 1997 report closed with two proposed extensions, the asymmetric case and a cost that depends on the pass, and with an observation that a working shift limits the work. All three are followed up here, with code, tests and measurements. This chapter follows them in turn, and for each problem states how hard it is, whether it is solved exactly or by a heuristic, and how far the heuristic can be from the optimum.

Starting point in 1997	Where it leads	Chapter
"La resolución de CPP para el caso asimétrico" (the asymmetric case)	Directed and mixed CPP	6.3, 9
"[...] que el valor asociado en cada arista variara en función de que fuera o no la primera vez que se recorre" (a different cost for the first pass)	Service and deadheading costs; the rural postman problem	10.2, 10.3
"[...] el trabajo que se puede realizar en media jornada [...] es claramente limitado" (a shift limits the work)	Several postmen; capacitated fleets	10.4, 10.5

10.1 Exact, heuristic, bound and guarantee

A problem is polynomial when an algorithm is known whose running time grows like a power of the network size; such problems can be solved exactly on city-sized networks. The undirected and directed CPP are polynomial. A problem is NP-hard when it is at least as hard as every problem in NP, the class of problems whose solutions can be checked in polynomial time: a polynomial algorithm for one NP-hard problem would give one for all of them, and none is known. For these there are two complementary methods. An exact method, here branch-and-cut, finds the optimum and proves it, but its running time can grow very fast. A heuristic answers quickly and is paired with a lower bound, a value that no solution can beat; the distance between the heuristic and the bound is the most the heuristic can be off. Some heuristics also carry a guarantee: a proof that they are never more than a fixed factor above the optimum, on any network.

Whether a polynomial algorithm can exist for the NP-hard problems is the P versus NP question, posed formally by Cook (1971) and, independently, by Levin (1973). It is one of the seven Millennium Prize Problems announced by the Clay Mathematics Institute in 2000, each with a prize of one million dollars; of the seven, only the Poincaré conjecture has been solved, and P versus NP remains open.

If P = NP were proved, every NP-hard problem in this chapter, and the mixed CPP of chapter 9, would have an exact polynomial algorithm, and heuristics would no longer be needed in principle. Two caveats apply: the proof might show that such an algorithm exists without showing how to build it, or it might give one whose running time grows with so high a power of the network size that it is useless in practice. Much of today's cryptography, which relies on some problems being hard, would also be at risk. If P ≠ NP, as most researchers expect, there is no shortcut, and the approach proposed in 1997 and followed here is the right one: an exact search as far as it can go, and beyond that a heuristic with a bound.

The postman shows how thin the line is. Edmonds' method solves the Chinese Postman Problem in polynomial time, yet mixing one-way with two-way streets, leaving some streets without service, or adding a second postman is enough to cross it.

10.2 First and later passes

Let every street e carry a service cost s(e), paid on the pass that serves it, and a travel cost c(e), paid on any other pass. The 1997 examples were a road covered in snow and a street whose bins are still full.

When every street needs service, each is served exactly once, so the service costs add up to a constant, and only the extra passes can change. These must make every degree even, as in chapter 3, and they cost c. Hence

$$\text{cost} = \sum_{e \in E} s(e) + \min_{\text{T-join } J} c(J),$$

which is the ordinary CPP augmentation computed on the travel costs. The problem stays polynomial: matching for undirected networks and minimum-cost flow for directed ones (`rural.solve_two_costs`). For mixed networks it inherits the NP-hardness of chapter 9.

10.3 The rural postman problem

When only a subset R of the streets needs service, the other streets may still be used to travel. This is the rural postman problem (RPP). It is NP-hard (Lenstra and Rinnooy Kan, 1976), but the difficulty is precise. If the required streets, together with the depot, form one connected piece, the argument of section 10.2 still applies: pair the odd vertices of R by shortest paths through the whole network, and the result is optimal in polynomial time. When R falls into several pieces, choosing how to join them is the hard part.

All three methods work on a reduced graph whose vertices are the endpoints of required streets and the depot. Two of them are joined by a deadhead edge whose cost is their shortest-path distance; edges whose shortest path passes through a third required vertex are left out, since they are never needed. With x(e) the number of extra passes over deadhead edge e, the model is

$$\min \sum_{e} c(e)\,x(e) \quad \text{s.t.} \quad x(\delta(v)) \equiv \deg_R(v) \pmod 2 \ \ \forall v, \qquad x(\delta(S)) \geq 2 \ \ \forall S \text{ union of pieces, } S \neq V, \qquad x$$

- **Heuristic** (`rural.heuristic`). Frederickson (1979): join the pieces by a minimum spanning tree of their distances, then pair the odd vertices by a minimum matching. It is never more than 3/2 times the optimum (Benavent, Campos, Corberán and Mota, 1985). The route is then improved by moving each street to its best position and by reversing stretches of the route (2-opt), working on the sequence of served streets (`sequence.improve`).
- **Lower bound** (`rural.lower_bound`). The best of two: the odd vertices of R must be paired, so the deadheading costs at least a minimum matching of them; and the linear relaxation of the model, with the cut around every piece and further cuts found by minimum cuts of the fractional solution.
- **Exact method** (`rural.solve_exact`). HiGHS solves the integer program; whenever its solution falls apart, the cut around each component is added and the program solved again. This is the model of Ghiani and Laporte (2000), with the cuts added one round at a time.

`rural.solve_rural` combines them as `solve_mixed` does in chapter 9: heuristic and bound first, then branch-and-cut told to beat the heuristic; if it cannot, the heuristic route is proven optimal.

Networks	Count	Streets to serve	Optimum proved	Frederickson above the optimum (mean, worst)	After local search (mean, worst)	Slowest proof
Benchmark files, required streets in one piece	51	28–347	51, by matching	—	—	3.7 s
Benchmark files, several pieces	6	51–159	6	0.81%, 3.23%	0.03%, 0.18%	6.2 s
Grid towns 10×10 and 20×20, 20% or 50% to serve	12	30–387	10	3.73%, 8.33%	1.04%, 4.75%	74 s

Networks	Count	Streets to serve	Optimum proved	Frederickson above the optimum (mean, worst)	After local search (mean, worst)	Slowest proof
Grid towns 30×30, 20% or 50% to serve	6	330–870	0 in two minutes	—	within 6.39% of the bound	—

The benchmark files are those of chapter 7 whose demands mark only some streets for service (Eglese and Beullens et al.); the towns are grids in which each street needs service with probability 0.2 or 0.5, with a two-minute limit. Frederickson's heuristic stays far inside its 3/2 guarantee: its worst case, on a small town with 15 pieces, is 8.3% above the optimum. The local search removes most of that excess, and branch-and-cut then either proves the route optimal or improves it. On the six largest towns, with 48 to 173 pieces, the optimum is not proved in two minutes, and the bound certifies the route within 0.8% to 6.4%.

10.4 Several postmen

When a district is too large for one shift, k postmen share it, all starting and ending at the depot. The min-max k-postmen problem asks for k routes that together serve every required street, with the longest route as short as possible. It is NP-hard even when every street needs service (Frederickson, Hecht and Kim, 1978).

- **Heuristic** (`fleet.solve_k_postmen`). Write out one postman's route as a sequence of served streets, then cut it into k consecutive pieces. A dynamic programme finds the cut points that make the longest route shortest; each piece becomes a route from the depot and back. Several written-out orders of the same route are tried. A local search then moves and swaps streets between the longest route and the others, re-plans each route as a small rural postman problem, and perturbs the best solution a few streets at a time while time remains.
- **Guarantee.** Let L be the length of an optimal single route, and r the largest cost of going from the depot to a required street, serving it and coming back. Any solution has a route at least r long, and the k routes together cost at least L, so the optimum is at least max(r, L/k). Frederickson, Hecht and Kim (1978) choose cut points on the optimal single route so that the longest route is at most (2 − 1/k) times the optimum. The dynamic programme considers every cut, so its result is at least as good; the guarantee holds whenever the single route is optimal, which is always the case when every street needs service.
- **Lower bound.** max(r, L/k), with L replaced by its lower bound when the single route is not proved optimal.

- **Exact method** (`fleet.solve_k_exact`), for small networks only: one copy of the rural model per postman, an assignment of every required street to one postman, the longest route as the objective, and connectivity cuts added round by round.

Networks	Count	Measure	Best split of one tour	After local search
Small random networks, 2 or 3 postmen	62	Above the exact optimum (mean, worst)	2.91%, 26.6%	0.02%, 1.25%
Small random networks	62	Optimal	—	61 of 62
Benchmark networks, 2 to 10 postmen	312 runs	Above the lower bound (median, worst)	12.0%, 44.6%	6.2%, 37.1%

On the small networks, half of them with every street to serve and half with 60% (two further cases with three postmen are left out, because the exact method did not finish within a minute), the best split of one tour is never more than 27% above the optimum, well inside the guarantee of 50% for two postmen and 67% for three, and the local search reaches the optimum in 61 of the 62 cases. On the benchmark networks the optimum is out of reach and only the lower bound is available, with 5 seconds of local search per run. A further 105 runs with more than 10 postmen are left out: there the farthest street sets the bound, and the longest route is at most 0.7% above it. Much of the remaining distance is the bound's: on the small networks, where the optimum is known, the same bound is a mean 3.0% and up to 17.5% below it. A stronger bound would be the natural next step.

10.5 Vehicles with a capacity

When each street produces a load and each vehicle carries a limited amount, the problem is the capacitated arc routing problem (CARP), the subject of the benchmark files used throughout this report. It is NP-hard (Golden and Wong, 1981). With the same sequence of served streets, Ulusoy's (1985) dynamic programme cuts the route into vehicle loads at least total cost; streets are then moved between routes for up to 5 seconds, while the total cost falls and the loads fit (`fleet.solve_carp`). The number of vehicles is left free.

On the 197 classical CARP instances (gdb, val, egl, kshs and the four Beullens families), the routes are a median 5.8% above the best solutions published (mean 5.5%, worst 14.8%), and match them on 16 instances; the split alone is a median 8.8% above. On 65 instances the routes use more vehicles than the file states, by up to 4, so those comparisons flatter the method: on the other 132 the median is 4.7%. 31 of the 197 best solutions are not proven optimal. The best

published solutions come from decades of dedicated metaheuristics and exact methods; the point here is how far the single tour of 1997 already goes when it is simply cut into vehicle loads.

10.6 Summary

Problem	Complexity	Exact method here	Heuristic and guarantee	Measured distance from the optimum
Every street, two-way or one-way	Polynomial	Matching or minimum-cost flow, in polynomial time	Not needed	0
Every street, first pass dearer	Polynomial	Matching on the travel costs	Not needed	0
Every street, mixed	NP-hard	Branch-and-cut, all benchmark networks and towns of up to 400 junctions; most up to 1,600	Orientation, balancing and local search	Mean 0.41%, worst 4.8%
Some streets (rural postman)	NP-hard; polynomial with one piece	Branch-and-cut, all 57 benchmark networks and 10 of the 12 towns of up to 400 junctions	Frederickson, at most 3/2; local search	Mean 0.66%, worst 4.8% (16 proved cases)
k postmen, min-max	NP-hard	Branch-and-cut, small networks	Best split, at most 2 − 1/k; local search	Mean 0.02%, worst 1.3% (62 small cases)
Capacitated fleet (CARP)	NP-hard	Not attempted	Ulusoy split and relocations	Median 5.8% above the best known

10.7 Still open

Some variants remain for future work: the windy postman, where a street costs more in one direction than the other (uphill, against the wind); turn penalties, where U-turns and left turns cost extra and which turn the problem into node routing on a larger graph; hierarchical service, where some streets must be served before others; and several depots.

Real street data. OpenStreetMap now provides free, worldwide street networks. The website of section 6.6 already reads them: it downloads a district's streets through the Overpass API, keeps junctions and street lengths, marks one-way streets and main roads, so a district such as the ones studied in 1997 can be modelled in seconds instead of digitised by hand. For whole towns, the OSMnx library (Boeing, 2017) downloads the same data as a NetworkX multigraph that `chinese_postman` takes after a short conversion.

11. Conclusions

The Chinese Postman Problem has been solvable in polynomial time since 1973, and today's implementations make that practical on any network a single service crew would plausibly cover. The 2026 version keeps the method of 1997 (pair the odd vertices by shortest paths, duplicate those paths, walk the Eulerian circuit) and improves every stage of it.

1. **Exact matching.** Edmonds' blossom algorithm, through NetworkX, gives the proven optimum on every instance tested: 4 classic graphs, 92 benchmark networks, 53 of them from real roads in the UK and Belgium, and 15 synthetic towns.
2. **Practical size.** With the Blossom V backend, a 40,000-junction network with 12,066 odd vertices solves in 102 seconds; in pure Python the same time covers about 800 odd vertices. Every benchmark network solves in under 2.5 seconds. Memory becomes the limit somewhere between the 12,066 odd vertices solved and the 19,942 that failed at 7 GB.
3. **Directed streets.** The directed CPP, proposed as future work in 1997, is solved by minimum-cost flow in under half a second on a 3,600-junction one-way grid.
4. **A bound for fleets that depends on capacity.** On 63 benchmark instances where every street needs service, the single-postman tour costs a median 95.1% of the best known multi-vehicle solution, from 100% down to 47.5%. The CPP is a quick estimate and a close lower bound when vehicles have capacity to spare, and a weak one when they do not.
5. **Mixed streets.** The mixed CPP, NP-hard, is solved exactly by branch-and-cut on every benchmark network (at most 5.8 s), on all 20 towns of up to 400 junctions and on 8 of the 10 of 900 and 1,600 junctions. Beyond that, five minutes of branch-and-cut end within about 1% of a proven bound, and a polynomial heuristic with dual-guided local search gives a tour 1% to 2.4% above that result in about a minute. Separation and bounding, the core of the 1997 work, is exactly what makes this possible.
6. **The paths left open in 1997.** A first pass dearer than the next keeps the problem polynomial when every street needs service. When only some streets do (the rural postman problem, NP-hard), branch-and-cut proves the optimum on all 57 benchmark networks that mark them, and the heuristic averages 0.66% above it. Several postmen sharing a district (NP-hard) are planned by the best split of one tour and a local search, optimal in 61 of 62 small cases where the optimum is known; with vehicle capacities, the same idea comes within a median 5.8% of the best published solutions.

7. **Open to everyone.** The code, the results, this report and the 1997 original are published at github.com/miquelmp/chinese-postman-problem, and the website solves street networks of up to 2,500 junctions in the browser, from a file or from OpenStreetMap, in seconds.

The two observations of 1997 about practice still hold. Routine services such as refuse collection or mail delivery are bounded by a working shift, so the networks involved stay moderate in size; and for one-off emergencies such as snow clearing, a fast near-optimal answer can be worth more than a slow proven one. For the CPP itself, thanks to the matching theory of the 1960s and 1970s and to today's implementations, a fast answer and a proven one are the same answer.

The natural continuation is a stronger lower bound for several postmen, the windy postman and turn penalties for real vehicles, all on the same code base, the same benchmark files and real street networks from OpenStreetMap.

References

Problem and algorithms

- Euler, L. (1741). Solutio problematis ad geometriam situs pertinentis. *Commentarii academiae scientiarum Petropolitanae* 8, 128–140.
- Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik* 1, 269–271.
- Kwan Mei-Ko (1960). Graphic programming using odd or even points. *Acta Mathematica Sinica* 10, 263–266 (in Chinese). English translation in *Chinese Mathematics* 1 (1962), 273–277.
- Edmonds, J. (1965). Paths, trees, and flowers. *Canadian Journal of Mathematics* 17, 449–467.
- Edmonds, J. (1965). Maximum matching and a polyhedron with 0,1-vertices. *Journal of Research of the National Bureau of Standards* 69B, 125–130.
- Edmonds, J. and Johnson, E. L. (1973). Matching, Euler tours and the Chinese postman. *Mathematical Programming* 5, 88–124.
- Hierholzer, C. (1873). Über die Möglichkeit, einen Linienzug ohne Wiederholung und ohne Unterbrechung zu umfahren. *Mathematische Annalen* 6, 30–32.

- Papadimitriou, C. H. (1976). On the complexity of edge traversing. *Journal of the ACM* 23(3), 544–554.
- Frederickson, G. N. (1979). Approximation algorithms for some postman problems. *Journal of the ACM* 26(3), 538–554.
- Cook, S. A. (1971). The complexity of theorem-proving procedures. *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing*, 151–158.
- Levin, L. A. (1973). Universal sequential search problems. *Problemy Peredachi Informatsii* 9(3), 115–116 (in Russian).
- Clay Mathematics Institute (2000). The Millennium Prize Problems: P vs NP. claymath.org/millennium/p-vs-np

Matching algorithms

- Gabow, H. N. (1973). *Implementations of algorithms for maximum matching on nonbipartite graphs*. PhD thesis, Stanford University.
- Lawler, E. L. (1976). *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart and Winston, New York.
- Micali, S. and Vazirani, V. V. (1980). An O(√|V|·|E|) algorithm for finding maximum matching in general graphs. *Proceedings of the 21st Annual Symposium on Foundations of Computer Science*, 17–27.
- Galil, Z., Micali, S. and Gabow, H. (1986). An O(EV log V) algorithm for finding a maximal weighted matching in general graphs. *SIAM Journal on Computing* 15(1), 120–130.
- Gabow, H. N. (1990). Data structures for weighted matching and nearest common ancestors with linking. *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms*, 434–443.
- Gabow, H. N. and Tarjan, R. E. (1991). Faster scaling algorithms for general graph matching problems. *Journal of the ACM* 38(4), 815–853.
- Cook, W. and Rohe, A. (1999). Computing minimum-weight perfect matchings. *INFORMS Journal on Computing* 11(2), 138–148.
- Mehlhorn, K. and Schäfer, G. (2002). Implementation of O(nm log n) weighted matchings in general graphs: the power of data structures. *ACM Journal of Experimental Algorithmics* 7, article 4.
- Kolmogorov, V. (2009). Blossom V: a new implementation of a minimum cost perfect matching algorithm. *Mathematical Programming Computation* 1(1), 43–67.

- Duan, R. and Pettie, S. (2014). Linear-time approximation for maximum weight matching. *Journal of the ACM* 61(1), article 1.
- Duan, R., Pettie, S. and Su, H.-H. (2018). Scaling algorithms for weighted matching in general graphs. *ACM Transactions on Algorithms* 14(1), article 8.
- Chen, L., Kyng, R., Liu, Y. P., Peng, R., Probst Gutenberg, M. and Sachdeva, S. (2022). Maximum flow and minimum-cost flow in almost-linear time. *Proceedings of the 63rd IEEE Symposium on Foundations of Computer Science*, 612–623.
- Wu, Y. and Zhong, L. (2023). Fusion Blossom: fast MWPM decoders for QEC. *Proceedings of the IEEE International Conference on Quantum Computing and Engineering*. arXiv:2305.08307.
- Higgott, O. and Gidney, C. (2025). Sparse Blossom: correcting a million errors per core second with minimum-weight matching. *Quantum* 9, 1600.
- Arkhipov, P. and Kolmogorov, V. (2026). Blossom VI: a practical minimum weight perfect matching algorithm. arXiv:2604.20351.

Arc routing

- Corberán, Á. and Laporte, G., eds. (2014). *Arc Routing: Problems, Methods, and Applications*. SIAM, Philadelphia.
- Eiselt, H. A., Gendreau, M. and Laporte, G. (1995). Arc routing problems, part I: the Chinese postman problem. *Operations Research* 43, 231–242.
- Lenstra, J. K. and Rinnooy Kan, A. H. G. (1976). On general routing problems. *Networks* 6(3), 273–280.
- Frederickson, G. N., Hecht, M. S. and Kim, C. E. (1978). Approximation algorithms for some routing problems. *SIAM Journal on Computing* 7(2), 178–193.
- Benavent, E., Campos, V., Corberán, Á. and Mota, E. (1985). Análisis de heurísticos para el problema del cartero rural. *Trabajos de Estadística y de Investigación Operativa* 36, 27–38.
- Golden, B. L. and Wong, R. T. (1981). Capacitated arc routing problems. *Networks* 11(3), 305–315.
- Ulusoy, G. (1985). The fleet size and mix problem for capacitated arc routing. *European Journal of Operational Research* 22(3), 329–337.
- Ghiani, G. and Laporte, G. (2000). A branch-and-cut algorithm for the undirected rural postman problem. *Mathematical Programming* 87(3), 467–481.

Benchmark instances

- Golden, B. L., DeArmon, J. S. and Baker, E. K. (1983). Computational experiments with algorithms for a class of routing problems. *Computers & Operations Research* 10, 47–59. (gdb)
- Benavent, E., Campos, V., Corberán, Á. and Mota, E. (1992). The capacitated arc routing problem: lower bounds. *Networks* 22, 669–690. (val)
- Li, L. Y. O. and Eglese, R. W. (1996). An interactive algorithm for vehicle routeing for winter-gritting. *Journal of the Operational Research Society* 47, 217–228. (egl)
- Beullens, P., Muyldermans, L., Cattrysse, D. and Van Oudheusden, D. (2003). A guided local search heuristic for the capacitated arc routing problem. *European Journal of Operational Research* 147, 629–643. (C–F)
- Brandão, J. and Eglese, R. (2008). A deterministic tabu search algorithm for the capacitated arc routing problem. *Computers & Operations Research* 35, 1112–1126. (egl-g)
- Martinelli, R. CARPData.jl: classical CARP instances in a common format. github.com/rafaelmartinelli/CARPData.jl (MIT licence; files used as distributed).
- Kiuchi, M., Shinano, Y., Hirabayashi, R. and Saruwatari, Y. (1995). An exact algorithm for the capacitated arc routing problem using parallel branch and bound method. *Abstracts of the 1995 Spring National Conference of the Operations Research Society of Japan*, 28–29. (kshs)

Software

- Hagberg, A., Schult, D. and Swart, P. (2008). Exploring network structure, dynamics, and function using NetworkX. *Proceedings of SciPy 2008*, 11–15.
- Virtanen, P. et al. (2020). SciPy 1.0: fundamental algorithms for scientific computing in Python. *Nature Methods* 17, 261–272.
- Huangfu, Q. and Hall, J. A. J. (2018). Parallelizing the dual revised simplex method. *Mathematical Programming Computation* 10, 119–142. (HiGHS)
- Dezső, B., Jüttner, A. and Kovács, P. (2011). LEMON – an open source C++ graph template library. *Electronic Notes in Theoretical Computer Science* 264(5), 23–45.
- Van Rantwijk, J. (2008). Maximum weighted matching in general graphs (mwmatching.py), jorisvr.nl/article/maximum-matching. Basis of the JavaScript matching code.
- Boeing, G. (2017). OSMnx: new methods for acquiring, constructing, analyzing, and visualizing complex street networks. *Computers, Environment and Urban Systems* 65, 126–139.

- Pyodide: Python compiled to WebAssembly, pyodide.org. Used for the in-browser optimality proof.
- OpenStreetMap contributors. Street data © OpenStreetMap contributors, Open Database Licence, read through the Overpass API.

Original project

- Monserrat Pons, M. (1997). *Resolución del problema del cartero chino (CPP) mediante la aplicación de un proceso de branch and bound*. Final-year project, ETSEIB, Barcelona. Supervisor: R. Companys Pascual.