P4: Delivery Routing with Learned Travel Times

Intelligent Systems course project. Team: Moaz Hassan El-Garawany (ID 320240032, AID S1) and Ahmed Hossam El-din (ID 320240051, AID S1). Date: 11 November 2026.

Project website: https://p4-delivery-routing.vercel.app (presentation: /slides.html).

All numbers in this report come from the committed code run on the machine used to build the project (`make_data.py`, `train_nn.py`, `run_experiments.py --seeds 10`, `run_demo.py`). The raw tables are in `report/results/` and the figures in `report/figures/`.

1. Problem

A dispatcher has about 20 deliveries for one weekday. Each one has a delivery zone, a time window, a demand and a service time. Three vans with different shifts and limited capacity are available. The system has to answer three questions:

1. **How long does a trip take** between two city zones at a given hour? We learn this from real NYC taxi trips with a neural network.
2. **Which van takes which order?** This is a constraint satisfaction problem (CSP): one variable per order, the domain is the 3 vans.
3. **In what order does each van visit its stops?** This is a search problem: A*, with UCS and Greedy as baselines.

The three parts work in a loop. If the search finds that a van cannot be on time, the CSP gets a new rule and re-assigns the orders. If no plan exists, the system says which orders cannot be served and why, and how much wider their windows would need to be.

Real vs generated data. The trip durations are real (NYC TLC, January 2022). The orders, windows, demands, service times, van capacities and shifts are generated by a seeded generator (`docs/data_dictionary.md`). Taxi times only approximate van times, so we multiply all predictions by a documented **van factor of 1.2**, in planning and in the replay alike.

2. Data

One month (January 2022, 38 MB parquet) is enough and kept the disk use small. After the filters (weekdays, duration 1-120 min, distance > 0, speed <= 60 mph), 1.72 M trips remain. We keep trips whose two ends are both in the 20 busiest zones (670 k trips) and sample **300,000** with seed 42. Nineteen zones are in Manhattan; the twentieth is **JFK Airport**, which gives the problem a few long, hard legs (about 40-50 taxi minutes).

The split is **by day**, never by row: train Jan 3-19 (173,350 trips), validation Jan 20-21 (33,058, used only for early stopping), test Jan 24-31 (93,592). The typical zone-pair distance, which is an NN input, is computed on the train days only.

Warnings from the specification, and what we found:

- The zone-ID columns exist in this month (checked automatically).
- Rare pairs at night are real: 8,406 of the 9,600 (pickup, drop-off, hour) cells appear in train, and 2,026 of them have fewer than 5 trips.
- The month has a handful of mis-dated records (52 outside January) and impossible speeds, which the filters remove.

3. Design choices

3.1 Neural network (`delivery/nn.py`)

The network is a hand-written numpy multi-layer perceptron (MLP), 73 -> 64 -> 32 -> 1 with ReLU.

- **Inputs:** one-hot pickup zone and one-hot drop-off zone; their first-layer weights act as learned 64-dimensional zone embeddings. Then one-hot hour plus sin/cos of the hour, one-hot weekday, and the zone-pair mean distance (standardised, plus its logarithm).
- **Leakage:** the trip's actual distance is not an input. It is unknown when planning.
- **Loss and training:** Huber loss (delta = 1 on the standardised target), so the model is robust to the long-tail trips that the 120-minute cap still allows. Adam (lr 1e-3, batch 512), L2 weight decay 1e-5, early stopping on validation MAE with patience 5.
- **Correctness:** backpropagation is checked against finite differences in the test suite (maximum relative error < 1e-5).

Training stopped at epoch 14; the best validation MAE was 2.59 min, at epoch 9, and those weights are kept. Each epoch takes about 0.4 s on a laptop CPU.

Baselines (`delivery/baselines.py`):

- **Historical average:** the mean duration per (pickup, drop-off, hour) on train. When a cell has fewer than 3 trips it backs off to the pair mean, then to the global mean.
- **Distance-linear:** least squares, duration = 6.03 + 1.90 x pair-distance (miles). It has no time-of-day information.

3.2 Time matrix and FIFO (`delivery/travel_time.py`)

Each model produces a matrix M[weekday, hour, i, j]. The planner reads it for the departure time t by linear interpolation between hour mid-points. As a result the travel time is continuous, and arriving later never helps ("first in, first out", FIFO), provided neighbouring hours differ by less than 60 min. This condition is checked for all three models.

FIFO plus waiting matters for the search: for the same (stop, visited set), an earlier clock time is always at least as good.

3.3 Search (`delivery/search.py`)

- **State:** (current stop, visited set, clock).
- **Action:** drive to an unvisited stop, wait if early, serve it.
- **Cost:** g = clock - shift start, which is the predicted travel plus waiting plus service time. Service time is a constant, so this equals "travel + waiting" up to a constant. With the return-trip extension the final leg back to the depot is included.
- **Windows:** in hard mode, arriving after a window closes prunes the successor.
- **Duplicates:** per (stop, visited set) we keep only non-dominated labels. With FIFO this is exact.
- **Re-opening:** A* re-opens a state when it is reached more cheaply, so admissibility alone guarantees optimality.

The heuristic and why it is admissible. The specification suggests "the sum of the cheapest predicted arrival into each unvisited stop" and asks us to check it.

- **Read naively, it is not admissible.** The naive reading uses the cheapest edge *at the current hour*. If traffic gets lighter later, or the van must wait for a window anyway and can then leave in lighter traffic, the remaining route can be cheaper than this estimate.
  `test_naive_current_hour_heuristic_can_overestimate` builds such a case: the naive heuristic says 40 min, the true optimum is 35.

- **Our fix** uses LB[u, k] = min over **all departure times >= now** of the predicted time u -> k, computed from a suffix minimum over the hourly matrices. Then:
    - **h_sum** = sum over unvisited k of (min over u in {current} ∪ unvisited {k} of LB[u, k]
        - service_k), plus the cheapest LB back to the depot. Every unvisited stop is entered by exactly one distinct leg, each leg costs at least its LB, and waiting is >= 0, so h_sum never exceeds the true remaining cost.
    - **h_wait** = max over unvisited k of (max(e_k, now + min-in_k) + service_k) - now, plus the return LB. A stop cannot be served before its window opens, so this is also a lower bound, and it is the part that "sees" waiting.
    - **h = max(h_sum, h_wait)**, which is admissible as the maximum of two admissible heuristics.
- **Weaker variant:** h = max over k of the cheapest incoming edge ("max-incoming"), also admissible, is included for comparison.
- **Tests:** admissibility is checked on 90 random states against exhaustive cost-to-go, and A* is checked against brute force on 6 stops.

Greedy is greedy best-first search on the same h. **UCS** is A* with h = 0.

3.4 CSP (`delivery/csp.py`)

- **Variable:** the van for each order. **Domain:** {0, 1, 2}.
- **Unary constraint:** the window must be reachable from the depot by that van, leaving at its shift start, and the van must still be able to return before its shift ends. Because the shifts differ (08:00, 09:00, 10:00), this really removes values: an order that closes at 09:20 cannot go on the 10:00 van.
- **Binary "different van" rules from NN predictions:** orders a and b must use different vans if depot -> a -> b and depot -> b -> a both miss a window, using the predicted times and the earliest shift start. This is a necessary condition for every van.
- **Learned rules:** the integration loop adds further rules (Section 3.5).
- **Global constraint:** total demand <= capacity and number of stops <= 10.

Solvers (switchable):

- **BT:** chronological backtracking.
- **FC:** forward checking. It removes the chosen van from the domains of neighbouring orders and from any order that no longer fits in that van's remaining capacity or stop budget.
- **AC-3:** arc consistency on the different-van constraints, run before search and after every assignment (MAC) on top of FC.

Orderings:

- Variable order is static: either input order, or most-constrained-first (highest degree).
- Value order puts the least-loaded van first, which spreads the work.
- With the same orderings, all three solvers return the **same first solution**; a test checks this. They differ only in how much of the tree they visit.

Counters: assignments, backtracks (assignments undone), consistency checks (BT only), FC prunes and AC-3 prunes.

3.5 Integration (`delivery/integration.py`)

```
for loop in 1..5:
    CSP (BT+FC+AC-3) with NN rules + learned rules  -> assignment, or "no solution"
    A* per van (hard windows); if no on-time order exists, re-run in soft mode
       (lateness penalised 1000/min) to get the least-late route
    if every route is on time: done
    for each late route: (stop before the first late stop, first late stop) -> new learned rule
```

If the CSP has no solution, `explain_no_solution` reports, in this order:

1. orders whose windows no van can reach (with the arrival time for each van);
2. demand or stop totals that exceed capacity;
3. a clique of mutually incompatible orders larger than the number of vans;
4. otherwise, the single orders whose removal makes the CSP solvable.

`suggest_widening` then pushes the latest time of the reported orders back in 15-minute steps (up to 2 h) and re-plans until the plan is feasible.

3.6 Replay simulation (`delivery/eval.py`)

A finished plan keeps its stop sequences. Each leg i -> j departing at time t gets a duration drawn from **real test-week trips** with the same (pickup, drop-off, hour), multiplied by 1.2. Pools with fewer than 3 trips back off to hour ± 1, then to all hours of that pair. The van waits if early, and we count deliveries that start after their window closes. Each plan is replayed 30 times (50 in the demo).

4. Results

4.1 Neural network vs baselines (test week, 93,592 trips)

Model	MAE (min)	RMSE (min)	Bias (min)
NN (MLP)	**3.232**	5.441	-1.87
Historical (pair, hour) average	3.244	**5.424**	-1.61
Distance-linear	4.144	6.401	-1.63
Global mean	5.454	8.880	-1.31

MAE by time of day (minutes):

Model	Night 0-5	Morning 6-9	Midday 10-15	Evening 16-19	Late 20-23
NN	**2.19**	3.82	3.90	**2.69**	**1.99**
Historical average	2.97	**3.75**	**3.87**	2.70	2.08
Distance-linear	4.02	4.66	4.87	3.49	2.92

An honest reading of these results:

- Overall, the NN only **ties** the historical average: it is 0.01 min better on MAE and 0.02 min worse on RMSE. With 300 k trips over 400 zone pairs, the busy (pair, hour) cells already have plenty of data, and their mean is very hard to beat.
- The NN wins where data is thin. On the 1,659 test trips whose (pair, hour) cell had fewer than 3 training trips, its MAE is **3.25 vs 4.69** for the historical average (and 4.87 for distance-linear). The embeddings share information across hours and zones, and that is the real value of the model here.
- Both learned models clearly beat the distance-only model, which cannot see congestion: the midday peak in `figures/nn_mae_by_hour.png` is where it fails most.

- All models under-predict the test week by about 1.6-1.9 min (negative bias). Traffic in late January was slower than in early January. This is a distribution shift that a single-month training set cannot capture.

Figures: `figures/nn_learning_curve.png`, `figures/nn_mae_by_hour.png`, `figures/nn_pred_vs_actual.png`, and the hourly matrix `figures/time_matrix_nn.png` (08:00 vs 18:00 heatmaps and the mean over pairs by hour). The predicted mean trip rises from about 10 min at night to about 15-16 min between 08:00 and 15:00.

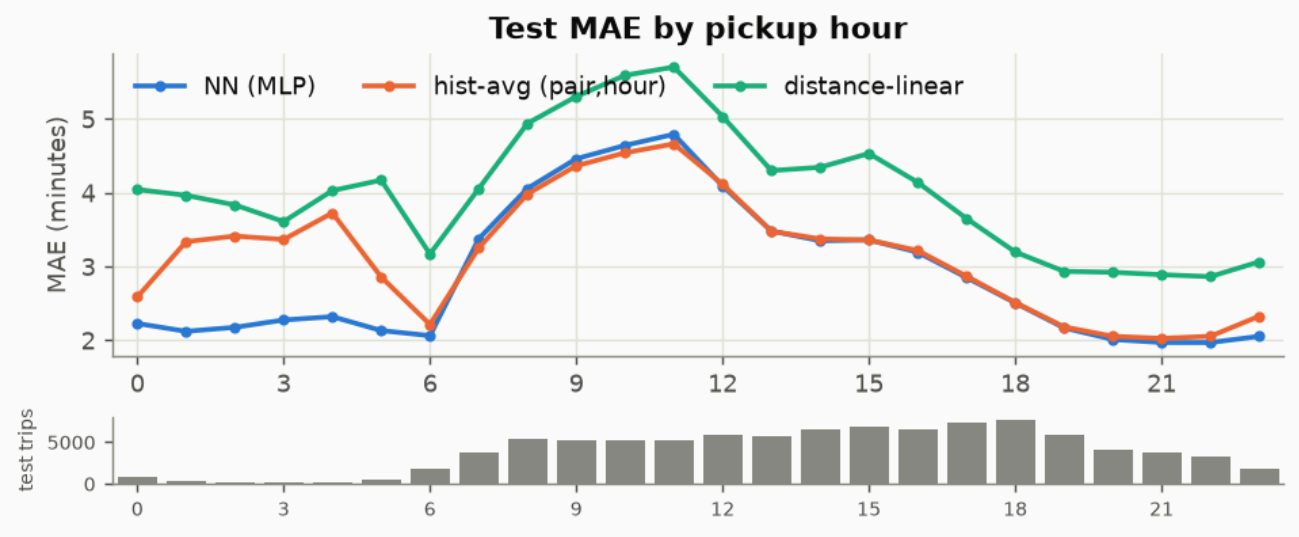


Figure 1. Test MAE by pickup hour for the NN and the baselines.

4.2 Experiment 1: A* vs UCS vs Greedy (6-10 stops, 10 seeds each)

One van, an instance that is feasible by construction, and NN times. Values are mean ± std over 10 seeds:

Stops	Algorithm	Expanded	Generated	Gap to optimum (%)	Time (ms)
6	UCS	95.5 ± 16.5	106.5 ± 22.2	0	0.60 ± 0.13
6	Greedy	64.7 ± 16.4	70.8 ± 16.2	6.7 ± 5.7	2.1 ± 0.5
6	A* (max-incoming)	51.3 ± 17.9	86.1 ± 19.2	0	1.9 ± 0.4
6	**A* (sum/wait)**	**34.0 ± 20.4**	63.7 ± 21.6	0	1.7 ± 0.7
8	UCS	356.7 ± 53.3	425.4 ± 75.2	0	2.7 ± 0.4
8	Greedy	326.4 ± 104	333.2 ± 105	3.7 ± 3.9	9.9 ± 3.1
8	A* (max-incoming)	213.8 ± 70.7	363.6 ± 58.9	0	8.1 ± 1.2
8	**A* (sum/wait)**	**109.3 ± 60.8**	227.4 ± 68.9	0	6.1 ± 2.0
10	UCS	1519 ± 161	1717 ± 301	0	13.4 ± 1.7
10	Greedy	1312 ± 440	1322 ± 441	4.6 ± 4.2	42.3 ± 14.8
10	A* (max-incoming)	865 ± 180	1439 ± 184	0	35.0 ± 5.4
10	**A* (sum/wait)**	**369 ± 249**	848 ± 293	0	24.7 ± 9.7

The rows for 7 and 9 stops are in `results/exp1_table.md`. A* matched brute force on all **30/30** instances with 6-8 stops. The plot is `figures/exp1_search.png`.

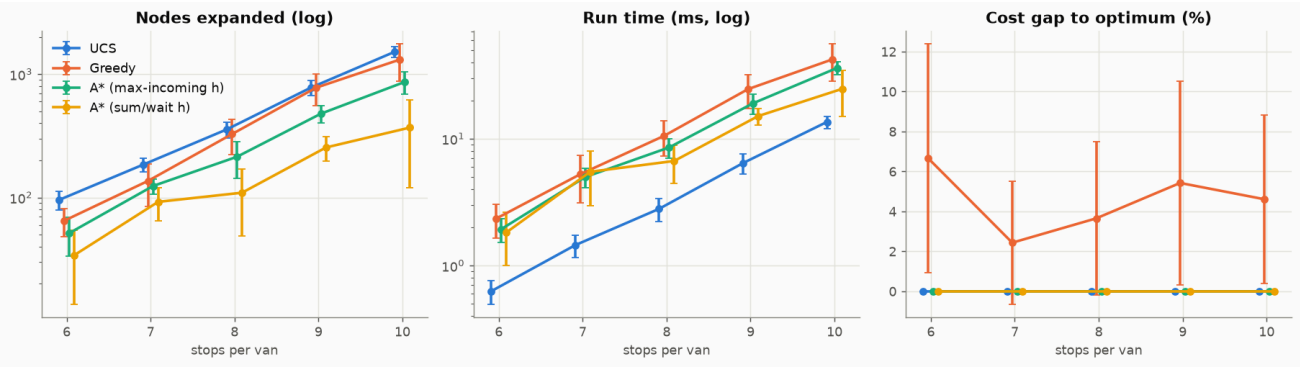


Figure 2. Experiment 1: nodes expanded, run time and cost gap, 6 to 10 stops.

- **A* is optimal and expands the fewest nodes:** 4.1x fewer than UCS at 10 stops. The stronger sum/wait heuristic expands 2.3x fewer nodes than max-incoming, which shows that a tighter admissible bound pays off.
- **But A* is slower in wall-clock time.** Each heuristic call costs $O(n^2)$ numpy work, and at ≤ 10 stops UCS's cheap expansions win (13 ms vs 25 ms). The gap in node counts grows quickly with n (UCS grows about 2.1x per extra stop, A* about 1.4-2.3x), so A* would win in time for larger routes.
- **Greedy is not a shortcut here.** Hard time windows create dead ends, so Greedy expands almost as many nodes as UCS, is the slowest, and its routes are 2-7% longer than optimal.

4.3 Experiment 2: BT vs BT+FC vs BT+FC+AC-3 (15/20/30 orders, 10 seeds each)

The instances are "rush hour": all windows fall between 09:00 and 12:00, are 15-45 min wide, and capacity has zero slack. They are generated with NN times; there are 2-8 NN "different van" rules per instance. The first three rows use input variable order; the fourth adds most-constrained-first order on top of FC+AC-3. Mean ± std, the median in brackets:

Orders	Solver	Assignments	Backtracks	Time (ms)	Solved / unsat / limit
15	BT	30.1 ± 28.4 [16.5]	15.1 ± 28.4	0.08	10/0/0
15	BT+FC	28.8 ± 27.4 [16]	13.8 ± 27.4	0.08	10/0/0
15	BT+FC+AC-3	28.6 ± 27.5 [15.5]	13.6 ± 27.5	0.38	10/0/0
15	+ degree order	15 ± 0 [15]	0	0.06	10/0/0
20	BT	446 ± 1230 [24]	426 ± 1230	1.16	10/0/0
20	BT+FC	192 ± 503 [23.5]	172 ± 503	0.47	10/0/0
20	BT+FC+AC-3	140 ± 349 [23]	120 ± 349	0.38	10/0/0
20	+ degree order	20.2 ± 0.4 [20]	0.2	0.12	10/0/0
30	BT	40,890 ± 79,600 [166]	40,870 ± 79,600	99.9	8/0/2
30	BT+FC	40,510 ± 79,800 [106]	40,490 ± 79,800	117.9	8/0/2
30	BT+FC+AC-3	22,030 ± 59,500 [105]	22,000 ± 59,500	66.1	8/1/1
30	+ degree order	4,475 ± 13,200 [37]	4,448 ± 13,200	11.6	9/1/0

The plot is `figures/exp2_csp.png`. "Limit" means the run stopped at 200,000 assignments.

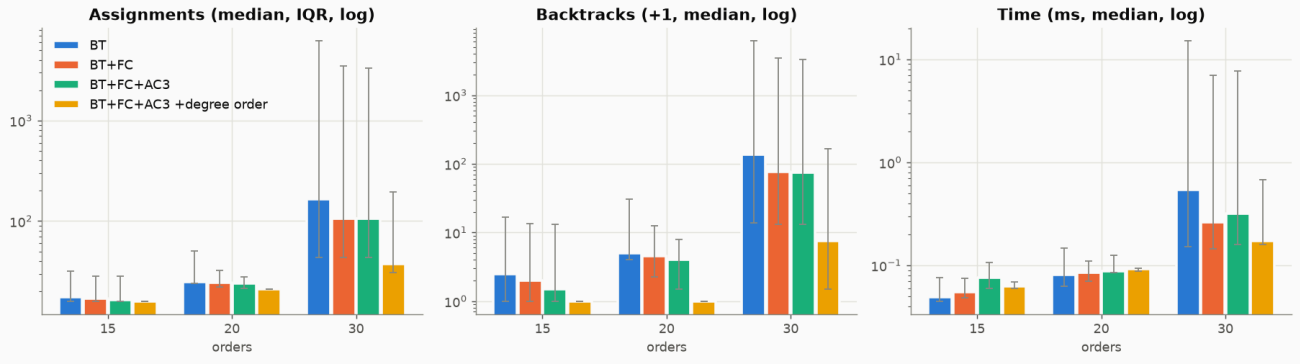


Figure 3. Experiment 2: BT vs BT+FC vs BT+FC+AC-3 for 15, 20 and 30 orders.

- **The distributions are heavy-tailed.** Most instances are easy, and one or two dominate the mean.
- **FC removes useless assignments.** It catches capacity dead ends early: median assignments drop from 166 to 106 at 30 orders, and the mean drops 2.3x at 20 orders.
- **AC-3 helps mainly on the hardest instance.** On 30-order seed 4, MAC proved infeasibility within the limit where BT and BT+FC ran out of budget.
- **Ordering matters even more than pruning.** Most-constrained-first order on top of FC+AC-3 solves 9/10 and proves the 10th infeasible, with a median of 37 assignments.
- **Why AC-3 adds little in general.** With 3 vans and only "≠" constraints, AC-3 can prune only when a neighbour's domain is already a single value. Most pruning power therefore comes from the capacity part of FC.
- **Small instances are trivial.** On the normal 15-20 order days (windows spread over 8 h) all solvers need only n assignments, as the demo shows.

4.4 Experiment 3: lateness in the replay (20 orders, 10 seeds x 30 replays)

The same instance is planned three times, with NN, historical-average and distance-linear times, and each plan is replayed with real test-week durations. Mean ± std over seeds:

Scenario	Times used for planning	Late deliveries / day	% late	Lateness (min/day)	Plans on time
Standard windows (30-120 min)	NN	0.31 ± 0.49	1.6 ± 2.4	7.8 ± 18.3	10/10
	historical average	0.39 ± 0.50	2.0 ± 2.5	7.0 ± 11.0	10/10
	distance-linear	0.31 ± 0.46	1.5 ± 2.3	5.1 ± 9.9	10/10
Tight windows (15-30 min)	NN	1.04 ± 0.69	5.2 ± 3.5	**9.1 ± 6.7**	10/10
	historical average	**1.01 ± 1.04**	**5.1 ± 5.2**	10.8 ± 12.7	10/10
	distance-linear	1.62 ± 1.21	8.1 ± 6.1	17.5 ± 15.1	9/10

The plot is `figures/exp3_lateness.png`.

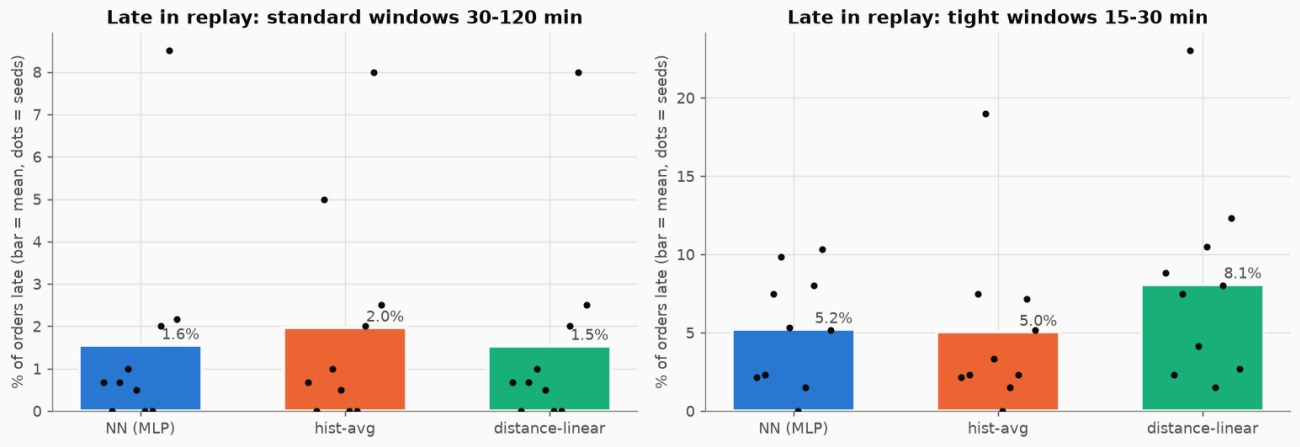


Figure 4. Experiment 3: late deliveries in the replay of real held-out trips.

- **Wide windows hide the time model.** There is enough slack that all three plans are almost never late, and the differences are within noise.
- **Tight windows reveal it.** The two models that know the hour (NN and historical) produce plans that are late about **35% less often** than the distance model's plans, with half the minutes of lateness.
- **NN vs historical is a tie in the replay too.** This is consistent with Section 4.1.
- **Our planned on-time rate is optimistic.** Every planned route was on time, yet 1-8% of deliveries are late in the replay. Two reasons: the predicted mean is not a guarantee, and the test week was slower than the training weeks.

4.5 The demo day (`python run_demo.py`, seed 4, Tuesday, depot Midtown Center)

- **Loop 1:** the CSP (2 NN rules) gives van 0 orders 16 (JFK 15:00-15:30) and 5 (Upper East Side North 15:45-16:30). A* finds no on-time order, and the least-late route is 2.1 min late. The pair (16, 5) becomes a learned rule.
- **Loop 2:** all 20 orders are on time. The vans return at 16:29, 16:38 and 15:49.
- **Search:** A* expanded 68 / 46 / 21 nodes for the three vans, against UCS's 132 / 130 / 64, with the same routes.
- **CSP:** on this day's constraint set, BT used 384 assignments (364 backtracks), FC 21 and FC+AC-3 20. On a hard rush-hour 20-order instance: 4,136 (BT) -> 1,701 (FC) -> 1,188 (FC+AC-3) -> 20 (+ degree order).
- **Replay (50 runs):**

Plan made with	Late deliveries per day
NN on	0.68
Historical average	1.38
NN off (distance)	0.70

A single instance is anecdotal, which is why Section 4.4 reports 10 seeds.

No-solution case (`python run_demo.py --impossible`): order 0 is moved to JFK with a window of 08:00-08:20. The report reads: *"order 0: window not reachable from the depot by any van: van 0: earliest arrival 08:48 is after the window closes 08:20; ..."* and suggests *"widening the windows of orders [0] by 30 min makes it feasible."*

Figures: `figures/demo_map.png` (routes on the zone map), `figures/demo_gantt.png` (arrivals vs windows), `figures/demo_constraint_graph.png` (NN rules in grey, the learned rule dashed red), `figures/demo_stats.png`.

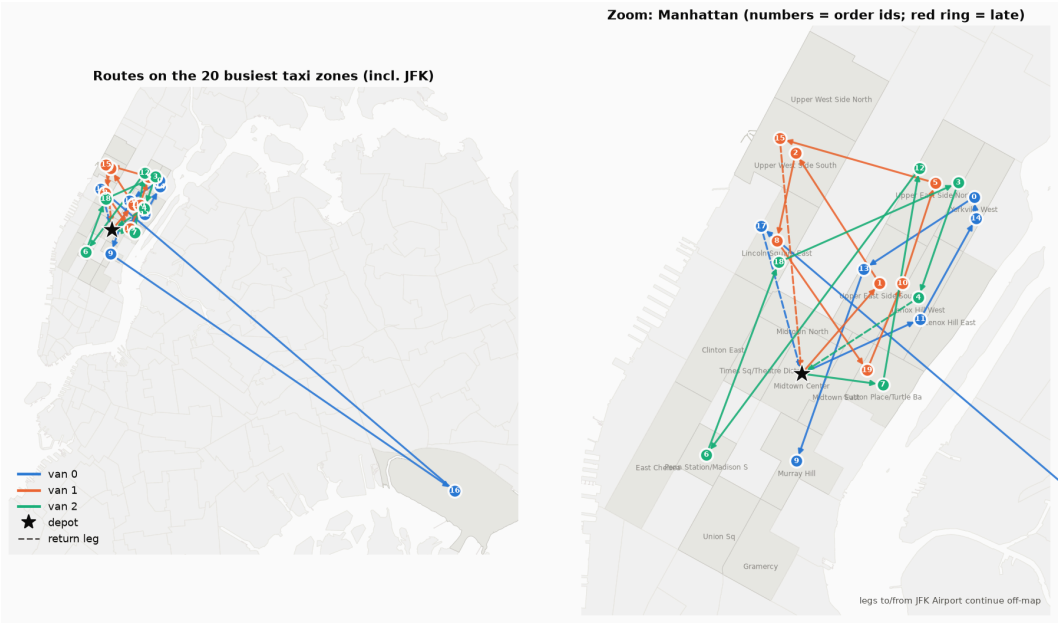


Figure 5. Demo day: the three van routes.

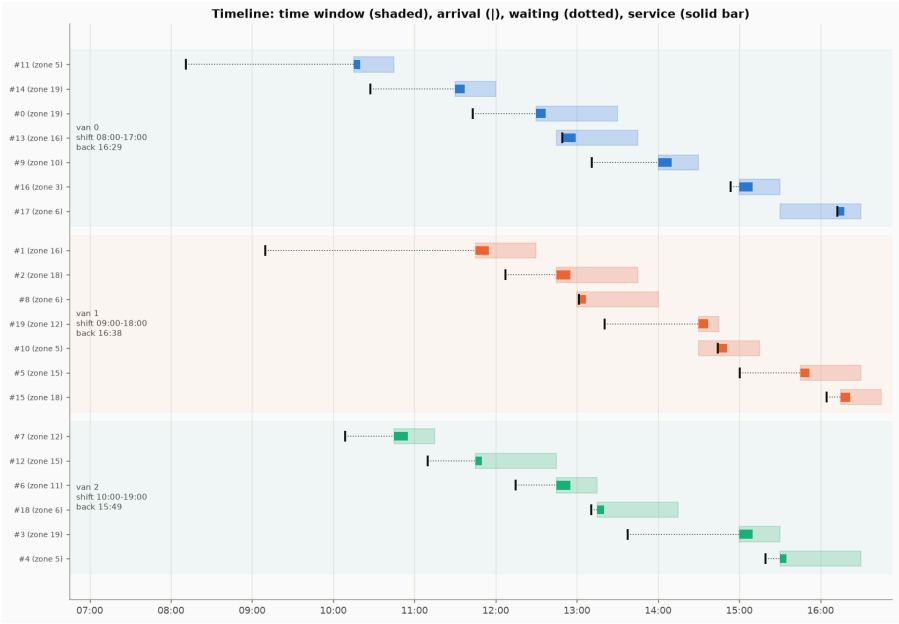


Figure 6. Demo day: time windows, arrivals, waiting and service per van.

4.6 Tests

`python -m pytest -q` runs **38 tests, all passing** in about 1-2 s on the committed fixture:

- **Normal cases (27):** a hand-computed 3-stop route (optimum C, B, A = 49 min); A* and UCS equal brute force on 6 stops for 5 seeds; heuristic admissibility; the naive heuristic overestimating; A* expanding fewer nodes than UCS; Greedy feasible but never better than optimal; the three CSP solvers valid and identical; FC never needing more assignments; AC-3 on a triangle; NN-derived conflict edges; integration serving every order once and on time; the replay counting lates; the backpropagation gradient check; NN training and save/load; the fixture's day split; seeded generator determinism; time interpolation and FIFO.
- **Edge cases (6):** an empty van, a stop in the depot zone, a zero-width window hit exactly, waiting when early, a late-shift van removed by the unary constraint, more vans than orders.
- **Impossible cases (5):** two incompatible stops for one van, four mutually conflicting orders with 3 vans, total demand above capacity, an unreachable window (with the widening suggestion), and the feedback

loop ending in "NO SOLUTION" for a single van.

5. Limits

- **Taxi is not van.** The factor 1.2 is an assumption, not a measurement. Vans also park, load and avoid some streets.
- **Mean predictions, hard windows.** The planner uses mean times with no safety buffer, so planned-on-time routes are late 1-8% of the time in the replay. A quantile model or a per-leg buffer would trade route length for punctuality.
- **One month, one city area.** January 2022 (Omicron-era traffic) and 20 zones. The test week was slower than the training weeks (bias -1.6 to -1.9 min), so retraining on more months would help.
- **The NN is not better than a lookup table on dense cells.** It helps on rare cells and gives a smooth, complete matrix.
- **The departure time is fixed at shift start.** Vans leave the depot at shift start, so early idle time shows up as waiting at the first stop. It does not change the stop order, but a real dispatcher would leave later.
- **The CSP ignores routing.** The CSP decides vans using pairwise rules only. Triples that do not fit are discovered by the search and fed back one pair per late route per loop, which needs up to 5 loops and is not guaranteed to converge to a feasible plan when one exists.
- **The search scales exponentially.** A* is exact and limited to about 10 stops per van, as in the specification. Larger vans would need a heuristic or local search.

6. Conclusions

The three techniques fit together naturally:

- The NN supplies a full hour-by-hour travel-time matrix (73 inputs, about 7k parameters).
- The CSP turns that matrix into rules ("these two orders cannot share a van") and solves the assignment.
- A* finds provably optimal stop orders. The admissible heuristic needed care because travel times depend on the hour and vans can wait.
- The feedback loop fixed the demo day in one extra iteration by learning one rule.

Experimentally:

- A* expands up to 4x fewer nodes than UCS.
- Forward checking, AC-3 and good variable ordering together reduce CSP work by orders of magnitude on hard instances.
- Planning with time-of-day-aware travel times reduces late deliveries by about a third compared with distance-based times when windows are tight.
- The learned NN is about as accurate as a well-filled historical lookup table, and more accurate where data is sparse.