

Implementation Instructions for Directed Rural Postman Support

This document describes the tasks required to extend the `feature/one-way` branch of **RppPrototype** so that the Rural Postman solver fully respects one-way restrictions not only for path lengths but also for the service edges themselves. The current branch provides a **directed driving graph** and an **undirected service graph**, so it already computes shortest paths that respect one-way streets. However, the required edges remain undirected and the solver still performs Christofides-style matching on an undirected multigraph. To implement a fully directed Rural Postman solver, follow the steps below.

1. Extend the graph loader

1. **Expose both directed and undirected representations:** The loader in `rpp/graph_loader.py` already generates a filtered `MultiDiGraph(G_filt)` and produces a directed or undirected driving graph based on the `ignore_oneway` flag ¹. Preserve this behaviour for the driving graph.
2. **Return a directed service graph:** Modify `load_graphs` to return three graphs:
3. `G_drive` – directed or undirected based on `ignore_oneway` (unchanged).
4. `G_service_undirected` – the current undirected graph used to obtain geometries (unchanged).
5. `G_service_directed` – a new `MultiDiGraph` obtained directly from the filtered `G_filt`. This graph will be used to build the required directed subgraph. It should keep edge attributes (e.g., `length`, `geometry`, `highway`) intact. You can simply assign `G_service_directed = G_filt` because `G_filt` is already a directed multigraph.

2. Build a directed required graph

`rpp/required_edges.py` currently constructs an undirected graph `R` of required edges by scanning `G_service` and adding any edge whose `highway` type is in `REQUIRED_HIGHWAYS` ². Replace `build_required_graph` with two functions:

- `build_required_graph_undirected(G_service_undirected) -> nx.Graph`: keep the existing behaviour for backward compatibility.
- `build_required_graph_directed(G_service_directed) -> nx.DiGraph`: iterate over the arcs `(u,v,key,data)` in `G_service_directed`. For each arc where the `highway` tag is in `REQUIRED_HIGHWAYS` and `is_driveable_edge(data)` is `True`, add a directed arc `(u,v)` to the required graph with the same weight. Do **not** add the reverse arc automatically – if a road is truly two-way in OSMnx, both directions should already be present in `G_service_directed`, so both will be added separately.

3. Implement a directed RPP solver

Create a new function `solve_drpp` in `rpp/rpp_solver.py`. The signature could be:

```

solve_drpp(
    G_drive: nx.MultiDiGraph,
    G_service_directed: nx.MultiDiGraph,
    R: nx.DiGraph,
) -> nx.MultiDiGraph

```

Follow these steps in the solver:

1. Connect required components:

2. Compute the strongly connected components of R . If there is only one component, skip this step. Otherwise, select a representative vertex from each component.
3. For each consecutive pair of components (C_i, C_{i+1}) , find a **shortest directed path** in G_{drive} from a representative of C_i to a representative of C_{i+1} . Use `nx.single_source_dijkstra` for this. If no directed path exists, try the reverse direction (as the one-way branch currently does). If neither direction exists, raise a `RuntimeError`. Store these connecting paths in a list.

4. Build the service multigraph:

5. Create an empty directed multigraph E (`nx.MultiDiGraph`).
6. For every directed required arc (u, v) in R , add it to E along with its geometry and weight from $G_{service_directed}$. Set a custom attribute `kind="required"`.
7. For each connector path, add each arc along the path to E with `kind="connector"`.

8. Compute vertex imbalances:

9. For each vertex v in E , calculate $\text{delta}[v] = \text{out_degree}(v) - \text{in_degree}(v)$. Maintain two lists: $D_{plus} = [v \text{ for } v \text{ in } E.\text{nodes}() \text{ if } \text{delta}[v] > 0]$ and $D_{minus} = [v \text{ for } v \text{ in } E.\text{nodes}() \text{ if } \text{delta}[v] < 0]$. If $\text{delta}[v] == 0$, the vertex is balanced. This corresponds to Thimbleby's polarity concept.

10. Construct the cost matrix:

11. For each i in D_{minus} and j in D_{plus} , compute the cost of the shortest directed path from i to j in G_{drive} . Use `nx.single_source_dijkstra` again. Store the path in a cache for later duplication. Paths may not exist due to one-way restrictions; skip any unreachable pairs.

12. Solve the min-cost flow problem:

13. Create a complete bipartite directed graph B with node set $D_{minus} \cup D_{plus}$. For each arc (i, j) where a path exists, assign cost equal to the shortest path weight and capacity equal to infinity (or a large integer). Define the supply/demand: for i in D_{minus} set `demand[i] = delta[i]` (negative), and for j in D_{plus} set `demand[j] = delta[j]` (positive).
14. Use `networkx.algorithms.flow.min_cost_flow` to solve the assignment: this will return a flow dictionary `f[i][j]` specifying how many times to duplicate the path from i to j . The

networkx min-cost flow implementation guarantees integer flows when capacities and demands are integer.

15. Duplicate balancing paths:

16. For each `i, j` where `f[i][j] > 0`, retrieve the cached shortest path between `i` and `j` and add all arcs along this path to `E`, repeating the addition `f[i][j]` times. Set `kind="duplicate"` on these arcs. This step balances the in- and out-degrees of every vertex.

17. Verify Eulerian property and compute tour:

18. After duplication, assert that every vertex now satisfies `out_degree == in_degree` and that `E` is strongly connected. Use `nx.is_eulerian(E)` for confirmation. Then call `nx.eulerian_circuit(E, keys=True)` to get a sequence of arcs forming the directed tour. In the GPX exporter, follow the geometry in the correct direction and reverse it when necessary (similar to the existing reversal logic).

19. **Return the Eulerian directed multigraph** `E`. Downstream tools can then export or visualise the route as before.

4. Modify the command-line interface and entry point

1. Add a new flag to `main.py` (e.g., `--directed-service`) that switches between the existing undirected solver and the new directed solver.
2. Use `load_graphs` to obtain `G_drive`, `G_service_undirected`, and `G_service_directed`.
3. Build the required graph by calling either `build_required_graph_undirected` or `build_required_graph_directed` based on the flag.
4. Invoke `solve_rpp` or `solve_drpp` accordingly.
5. Export the resulting tour with the existing GPX exporter (which must be updated to handle directed edges as mentioned in step 3.7).

5. Testing and validation

1. **Unit tests** – Create small directed graphs with known one-way restrictions and required arcs to verify that the directed solver respects arc directions and produces a valid Eulerian tour. Tests should also check that the solver raises errors when no directed path exists between required components.
2. **Regression tests** – Ensure that the undirected mode behaviour remains unchanged when the new flag is disabled. The one-way branch's existing tests for undirected RPP should continue to pass.
3. **Real OSM examples** – Run the solver on a small area with known one-way streets (e.g., a city block with a one-way loop) and inspect the generated tour to confirm that no street is traversed in the wrong direction.

Implementing the above changes will evolve the current one-way branch into a solver that truly supports directed service graphs. The design reuses the directed driving graph and geometry handling while adding a directed required graph, directed balancing via min-cost flow, and an Eulerian tour through a `MultiDiGraph`.

1 **graph_loader.py**

https://github.com/Pygmalion69/RppPrototype/blob/feature/one-way/rpp/graph_loader.py

2 **required_edges.py**

https://github.com/Pygmalion69/RppPrototype/blob/feature/one-way/rpp/required_edges.py