Multi-Vehicle Route Planning for Efficient Urban Freight Transport

Petar Mrazovic
Royal Institute of Technology, Sweden
mrazovic@kth.se

Elif Eser
Bilkent University, Turkey
elif.eser@bilkent.edu.tr

Hakan Ferhatosmanoglu
University of Warwick, UK
hakan.f@warwick.ac.uk

Josep L. Larriba-Pey
Polytechnic University of Catalonia
larri@ac.upc.edu

Mihhail Matskin
Royal Institute of Technology, Sweden
misha@kth.se

ABSTRACT

The urban parking spaces for loading/unloading are typically over-occupied, which shifts delivery operations to traffic lanes and pavements, increases traffic, generates noise, and causes pollution. We present a data analytics based routing optimization that improves the circulation of vehicles and utilization of parking spaces. We formalize this new problem and develop a novel multi-vehicle route planner that avoids congestions at loading/unloading areas and minimizes the total duration. We present the developed tool with an illustration and analysis for the urban freight in the city of Barcelona, which monitors tens of thousands of deliveries every day. Our system includes an effective evaluation of candidate routes by considering the waiting times and further delays of other deliverers as a first class citizen in the optimization. A two-layer local search is proposed with a greedy randomized adaptive method for variable neighborhood search. Our approach is applied and validated over data collected across Barcelona's urban freight transport network, which contains 3,704,034 parking activities. Our solution is shown to significantly improve the use of available parking spaces and the circulation of vehicles, as evidenced by the results. The analysis also provides useful insights on how to manage delivery routes and parking spaces for sustainable urban freight transport and city logistics.

1 INTRODUCTION

Higher consumption of goods and services increases the demand for urban freight distribution. According to European Environment Agency, urban freight demand has increased ~ 34% from 1995 to 2014 on road freight transport [1]. Urban deliveries, which are typically made in small loads and frequent runs, increase traffic and generate noise and pollutant emissions. Lack of enough loading/unloading areas shifts delivery operations to traffic lanes and pavements which leads to congestion and poses a threat to the safety of other road users [7].

The provision of dedicated loading[1] areas has been recognized as the most effective policy for organizing last-mile delivery operations [7]. The city of Barcelona also follows this common approach and controls the delivery parking spaces, allowing the vehicles to take a specific slot in real time in these areas, which are locally known as *AreaDUM* (Distribució Urbana de Mercaderies in Catalan). However, the available loading areas can not absorb rapidly increasing urban transport demands. The parking places are over-occupied and there is not much room left to improve the urban transport and parking infrastructure.

[1]We use "loading" in short while referring to both loading and unloading activities.

In this paper, we present our data science based solution to improve the planning of freight transport, in collaboration with the city of Barcelona. We analyze mobility data from Barcelona's regulated delivery areas, and develop a technology to improve the utilization of their loading/parking areas. Barcelona has around 9,000 parking spaces for freight deliveries across 2,200 areas that are designated for freight transport. There are thousands of vehicles visiting these areas every day which cause congestions both in the edges (roads) and in the nodes (loading areas).

We propose a novel approach of planning multiple vehicles' routes with a collective optimization task, as opposed to traditional solutions that merely compute individually optimal routes for each vehicle. We formally define the problem and mathematical model of multi-vehicle route planning, which is shown to be NP-complete. We present a variant of the Hamiltonian Path Problem (HPP) where each deliverer visits a set of loading areas while her route choice affects the delay of other deliverers due to the limited capacity of the areas. The waiting times at parking/loading areas are considered as a first-class citizen. This is a practical factor which is not addressed by traditional approaches that aim to minimize travel time only. Deliverers tend to seek routes that minimize their own travel time, which leads to congestions at the parking areas and thus additionally increases duration of delivery routes.

Our approach of targeting a global objective in identifying the set of delivery routes improves mobility and minimizes the time spent in the loading areas. It trades individual sub-optimal routes when they together minimize the overall aggregate travel costs, and avoids cascades of delays due to the limited capacities of loading areas. To efficiently compute the total time cost of a route arrangement including travel and waiting times, we use a priority queue structure to maintain arrivals and departures of vehicles. The queue waiting time at all loading areas, which depend on different delivery durations and routes in the system, is computed efficiently in $O(n\ log(n))$ time, with n being the total number of deliveries.

We analyze freight transport data collected from citizens and deliverers, and illustrate our system within the urban freight transportation in the city of Barcelona. The dataset contains 3,704,034 parking information for 49,172 deliverers. The results confirm that the multi-vehicle delivery planner improves the circulation of vehicles and avoids congestions at the loading areas. Our paper makes the case and show the need for a collective planning of urban freight transportation with a global optimization task and fairness to all deliverers in the system.

The rest of this paper is organized as follows. In the next section we define the proposed multi-vehicle route planning problem. In Section 3 we present our solution and discuss its design details.

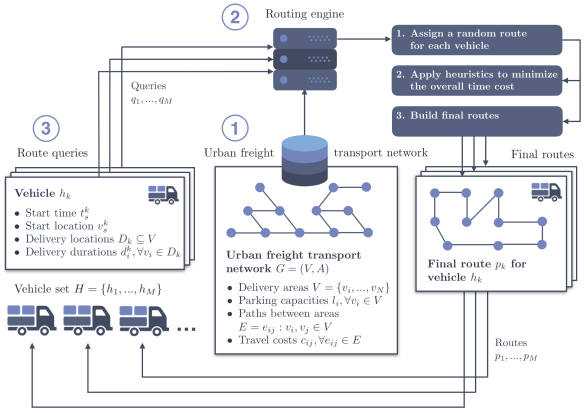

Figure 1: System components of the proposed multi-vehicle route planner

Section 4 presents our implementation and experimental results on data collected across Barcelona's urban freight transport network. In Section 5 we give an overview of the related work. Finally, we conclude and discuss future work in Section 6.

2 PROBLEM STATEMENT

In this section, we summarize the main requirements and architecture of the routing system, and define the underlying multi-route optimization problem.

As illustrated in Figure 1, the developed multi-route planning system consists of (1) urban freight transport network, (2) routing engine, and (3) route query set. The transport network represents the underlying physical traffic network composed of loading areas and roads between them. Naturally, the network can be formalized with the aid of directed graph $G(V, E)$ whose node set $V = \{v_1, ..., v_N\}$ represent loading areas and edge set $E = \{e_{ij} | v_i, v_j \in V\}$ shortest path between them. Furthermore, vehicle capacity l_i and travel cost c_{ij} are assigned to each node $v_i \in V$ and edge $e_{ij} \in E$, respectively. Urban freight transport network, i.e., its graph model, is stored in a graph database and accessed by the routing engine.

Definition 2.1 (Graph model). *Let $G(V, E)$ be a directed graph where $V = \{v_1, ..., v_N\}$ is the set of nodes representing loading areas, and $E = \{e_{ij} | v_i, v_j \in V\}$ is the set of edges representing the shortest paths between them. Each node $v_i \in V$ is assigned with the capacity l_i representing the number of parking spaces at the loading area v_i. Each edge $e_{ij} \in E$ is assigned with the travel cost c_{ij}.*

The routing engine is the central component that consists of our solutions for near optimal solving complex route optimization problems. It takes as input a set of route queries $Q = \{q_1, ..., q_M\}$ assigned to delivery vehicles $H = \{h_1, ..., h_M\}$. Each query q_k consists of the vehicle h_k's initial location v_s^k, delivery start time t_s^k, set of delivery locations (i.e., loading areas) to be visited along the planned route D_k, and estimated delivery durations $d_i^k \ \forall v_i \in D_k$.

Definition 2.2 (Route query). *Vehicle h_k's route query q_k is composed of its initial position $v_s^k \in V$, delivery start time t_s^k, set of predefined delivery locations $D_k \subseteq V$, and estimated delivery durations $\{d_i^k | v_i \in D_k\}$.*

Route queries $Q = \{q_1, ..., q_M\}$ are concurrent and their delivery location sets $D_k \ \forall q_k \in Q$ can intersect. The loading areas are limited in capacity, and thus can only accommodate limited number of vehicles at the same time. This means that the route choice for a particular vehicle affects the duration of routes planned for other vehicles in the system. In other words, if a delivery area is at its full capacity, other vehicles need to wait in a queue based on *first-come-first-served* policy before being parked at the congested area (Figure 2). Therefore, the goal is to build an optimal set of routes $P = \{p_1, ..., p_M\}$ (one for each query $q_k \in Q$) which collectively minimizes the total time cost in the transport system expressed as the sum of the durations of all the routes $p_k \in P$.

Definition 2.3 (Route duration). *The duration of a single route p_k is computed as the sum of the travel times along p_k, delivery durations $d_i^k \ \forall v_i \in D_k$, and waiting times $w_i^k \ \forall v_i \in D_k$. Formally,*

$$cost(p_k) = \sum_{v_i \in D_k} \left(d_i^k + w_i^k + \sum_{v_j \in D_k} x_{ij}^k c_{ij} \right), \tag{1}$$

where $x_{ij}^k \in \{0, 1\}$ is a decision variable defined to be equal to 1 if edge e_{ij} is traversed in route p_k, and equal to 0, otherwise.

Recall that for a particular route p_k, the travel costs c_{ij} and estimated delivery durations $d_i^k \ \forall v_i \in D_k$ are known from the graph model $G(V, E)$ and route query q_k, respectively. On the other hand, the queue waiting time w_i^k at each loading area $v_i \in D_k$ needs to be explicitly computed. However, such computation is not straightforward since it depends on other routes in the system, while the delivery durations differ from route to route. Therefore, in the next section we propose an efficient algorithm to compute the total waiting time in $O(n \ log(n))$ time.

We formally define the goal of the proposed multi-vehicle optimization problem as follows:

Definition 2.4 (Problem goal). *Given an urban freight transport network $G(V, E)$ and a fleet of vehicles $H = \{h_1, ..., h_M\}$ with the corresponding set of route queries $Q = \{q_1, ..., q_M\}$, the goal of the proposed problem is to determine the set of M routes $P = \{p_1, ..., p_M\}$ that collectively minimize the total time cost, while visiting all of the planned nodes $v_i \in D_k \ \forall q_k \in Q$. We further formalize it as follows,*

$$\begin{aligned} min \ f(P) &= \sum_{p_k \in P} cost(p_k) \\ &= \sum_{p_k \in P} \sum_{v_i \in D_k} \left(d_i^k + w_i^k + \sum_{v_j \in D_k} x_{ij}^k c_{ij} \right), \end{aligned} \tag{2}$$

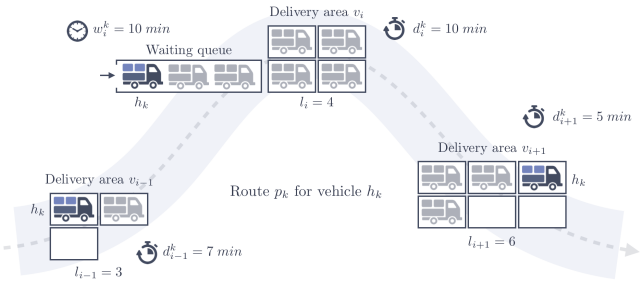

Figure 2: An example of a route with a congested loading area. At the time vehicle h_k arrived at v_i, the area was at its full capacity and thus h_k was placed in a waiting queue.

where x_{ij}^k is a decision variable introduced in Definition 2.3, and each route $p_k \in P$ visits all of the planned nodes $v_i \in D_k$ starting at node v_s^k and time t_s^k.

If the loading areas had unlimited capacities (and consequently no waiting queues), a relaxed variant of the proposed optimization problem can be reduced to a set of separate NP-complete HPPs [12]. The optimal solution, in this case, consists of a set of the shortest travel routes computed individually for each route query. In real life, deliverers tend to seek such routes in order to minimize their total travel time. However, in Section 4 we will show that such approach in busy freight networks often leads to congestions and waitings at the delivery areas which additionally increases duration of delivery routes. Therefore, in this paper we show that in dense urban environments delivery routes should be planned collectively in advance with fair respect to all deliverers in the system.

3 THE PROPOSED SOLUTION

We first introduce the concept of solution representation and propose an efficient algorithm to compute the objective function values (i.e., the total time cost of a solution). We then present the construction and improvement/search phases of the proposed solution.

3.1 Solution representation and objective function computation

A solution in our algorithm is represented as a combination of permutations of nodes in each path $p_k \in P$. Therefore, a single path $p_k \in P$ is represented as an ordered set (i.e., tuple) of nodes from delivery location list D_k. For example, a solution $P = \{p_1, p_2, p_3\}$ where $D_1 = \{v_1, v_2, v_3, v_4, v_5\}$, $D_2 = \{v_2, v_4, v_6, v_8\}$, $D_3 = \{v_1, v_5, v_6, v_7, v_9\}$, $v_s^1 = v_s^3 = v_1$, and $v_s^2 = v_2$, is represented as a combination of ordered sets, i.e.,

$$P = \left\{ \begin{array}{l} p_1 = (v_1, v_5, v_3, v_4, v_2), \\ p_2 = (v_2, v_6, v_8, v_4), \\ p_3 = (v_1, v_7, v_9, v_5, v_6) \end{array} \right\}. \quad (3)$$

From the introduced representation we can compute the objective function value. As duration of delivery differs from vehicle to vehicle, and from node to node, finding the order in which vehicles leave the waiting queues is not obvious. Therefore, we introduce the concept of *events* and develop a computationally inexpensive algorithm to compute the objective function values.

The proposed algorithm tracks arrivals and departures at/from loading areas. In Algorithm 1, each such event is modelled as a tuple containing the type of the event (arrival or departure), corresponding node and route, and the scheduled time. For example, tuple $(Arrival, p_k, v_i, t_i^k)$ represents arrival at node v_i in route p_k scheduled at time t_i^k. Similarly, tuple $(Departure, p_k, v_i, z_i^k)$ represents departure from node v_i in route p_k scheduled at time z_i^k. Notice that we use a different notation for a scheduled arrival time (t_i^k) and departure time (z_i^k). This is due to the fact that these times are correlated, meaning that one can be computed as a result of the other by knowing estimated waiting time, delivery duration and travel cost. We formally define the arrival and departure times, and discuss their relationship in Remark 3.1.

Definition 3.1 (Arrival/Departure time). *Arrival time t_i^k is the time at which vehicle h_k is scheduled to arrive at node v_i in path p_k. Departure time z_i^k is the time at which vehicle h_k is scheduled to departure from node v_i in path p_k.*

ALGORITHM 1: Objective function computation

Input: Solution $P = \{p_1, \ldots, p_M\}$
Output: Objective function value, i.e., total time cost

```
1  Procedure ComputeCost(P)
2      departures[v_i] ← [∅], ∀v_i ∈ V
3      cost[p_k] ← 0, ∀p_k ∈ P
4      T ← [∅]
5      forall p_k ∈ P do
6          T.enqueue(Arrival, p_k, v_s^k, t_s^k)
7      end
8      while T not empty do
9          e ← T.dequeue()
10         Let v_i represent the node assigned to event e
11         Let p_k represent the route assigned to event e
12         if event e is an arrival then
13             Let t_i^k represent the scheduled time of event e
14             if departures[v_i].length < l_i then
15                 z_i^k ← t_i^k + d_i^k
16             else
17                 b ← departures[v_i].length - l_i
18                 z_i^b ← departures[v_i].get(b)
19                 z_i^k ← z_i^b + d_i^k
20                 cost[p_k] ← cost[p_k]+z_i^k − t_i^k
21             end
22             departures[v_i].add(z_i^k)
23             T.enqueue(Departure, p_k, v_i^k, z_i^k)
24         else if event e is a departure then
25             Let z_i^k represent the scheduled time of event e
26             departures[v_i].remove(z_i^k)
27             if v_i is not the last node in p_k then
28                 Let v_j represent v_i's successor in p_k
29                 t_j^k ← z_i^k + c_ij
30                 cost[p_k] ← cost[p_k]+c_ij
31                 T.enqueue(Arrival, p_k, v_j, t_j^k)
32             end
33         end
34     end
35     return Σ_{p_k∈P} cost[p_k]
```

Remark 3.1 (Correlation between arrival and departure time). *Since vehicles depart from loading areas after finishing their deliveries, departure time z_i^k can be computed from arrival time t_i^k by adding waiting time w_i^k and delivery duration d_i^k. Formally,*

$$z_i^k = t_i^k + w_i^k + d_i^k. \quad (4)$$

On the other hand, arrival time t_i^k can be computed from departure time z_j^k from v_i's predecessor v_j, by adding travel cost c_{ji}, i.e.,

$$t_i^k = z_j^k + c_{ji}. \quad (5)$$

Before we discuss Algorithm 1 in details, we need to introduce the concept of *event queue*, which is a priority queue to store and process arrival and departure events.

Definition 3.2 (Event queue). *Event queue T is a priority queue containing arrival and departure events sorted by their scheduled times in ascending order. The queue is implemented as an array indexed by priority, where each array cell contains an event with a scheduled time as priority. By convention, we call the queue insert operation* enqueue *and the remove operation* dequeue.

Algorithm 1 uses the event queue as a buffer where scheduled events wait to be processed one by one. In other words, until event queue T is emptied, at each iteration, the algorithm will remove

and process the earliest scheduled event from T (lines 8-34). If the currently being processed event (e) is an arrival at node v_i in route p_k, the algorithm computes departure time z_i^k using (4), and inserts a new departure event into event queue T (lines 13-23). However, in order to compute a new departure time z_i^k we need to know the exact time at which delivery area v_i becomes available for vehicle h_k. For that purpose, for each node $v_i \in V$ we keep a list of departure times scheduled for the events contained in T that still need to be processed (`departures[`v_i`]` $\forall v_i \in V$). If the size of list `departures[`v_i`]` is larger than node v_i's capacity l_i, then vehicle h_k will be placed in the waiting queue. In this case, we compute the size of the waiting queue b (line 17) and retrieve the b-th earliest departure time z_i^b at which loading area v_i will become available for vehicle h_k from `departures[`v_i`]` (line 18). z_i^k is then computed from z_i^b by adding delivery duration d_i^k (line 19). In line 20, time cost `cost[`p_k`]` is increased by waiting time w_i^k and delivery duration d_i^k computed as the time difference between t_i^k and z_i^k. However, if event e is a departure from node v_i in route p_k and v_i is not the last node in p_k, the algorithm computes arrival time t_j^k at v_i's successor v_j using (5), and inserts a new arrival event into event queue T (lines 25-32). Time cost `cost[`p_k`]` is then increased by travel cost c_{ij} in line 30. Finally, the total time cost of solution P is computed in line 35 by the summarizing time costs across all the routes $p_k \in P$. The event queue is initialized at the beginning of Algorithm 1 in lines 4-7. For each route $p_k \in P$, the arrival at the initial node v_s^k is scheduled at delivery start time t_s^k and added to event queue T.

Remark 3.2 (Time complexity of Algorithm 1). *We assume that the event queue is implemented using a heap and the departure lists using self-balancing binary search trees. This means that an event can be enqueued and dequeued in $O(log(n))$, with n being the total number of events, and, similarly, a departure time can be inserted, removed and searched for in $O(log(n))$. Since Algorithm 1 needs to process two events (i.e., arrival and departure) for each node visited in P, its total time complexity is $O(n\ log(n))$.*

3.2 Construction phase

In the construction phase of the proposed solution, we build an initial solution P_0 in a greedy fashion. Here, we focus only on travel time and aim to minimize it for each individual route $p_k \in P$. In other words, we seek for near-optimal solutions to a set of HPPs. Greedy algorithms have the advantage of being fast and easy to implement. However, since the purpose of the initial solution is to position the search in a promising area of the solution space, a completely greedy algorithm can be expected to miss some potentially valuable areas. Therefore, in the construction phase we combine greediness and randomness using GRASP [10].

To obtain variability in the candidate set of greedy solutions, GRASP keeps the best-ranked solution elements in a restricted candidate list (RCL) from where some of them are randomly chosen when building up the solution. Usually, the size of RCL, α, is used as a parameter to control the balance between greediness and randomness. In othr words, when α is small, the algorithm becomes greedier, but when α is large, the algorithm becomes more random. We refer the interested reader to [23] for a more extensive description of GRASP.

ALGORITHM 2: The construction of initial solution

Input: Route query set $Q = \{q_1, \ldots, q_M\}$
Output: Initial solution $P_0 = \{p_1, \ldots, p_M\}$
1 **Procedure** ConstructRandomSolution(Q)
2 $P_0 \leftarrow \emptyset$
3 **forall** $q_k \in Q$ **do**
4 $p_k \leftarrow (v_s^k)$
5 $v_r \leftarrow v_s^k$
6 $V_u \leftarrow D_k \setminus v_r$
7 **while** V_u not empty **do**
8 $\alpha \leftarrow \lceil \alpha_g \cdot |V_u| \rceil$
9 $RCL \leftarrow \alpha$ nodes from V_u closest to v_r
10 $v_r \leftarrow$ random node from RCL
11 p_k.add(v_r)
12 $V_u \leftarrow V_u \setminus v_r$
13 **end**
14 $P_0 \leftarrow P_0 \cup \{p_k\}$
15 **end**
16 return P_0

The GRASP's concept of RCL can be translated into many combinatorial optimization problems. In order to obtain semi-greedy solutions to HPPs, we implement RCL using the distance matrix containing pre-calculated travel costs between each pair of loading areas $(v_i, v_j) \in V \bigtimes V$. For each path $p_k \in P$, starting from initial node v_s^k, RCL is populated with the α closest loading areas, i.e. nodes, using the pre-computed distance matrix. Afterwards, a random node v_r is chosen from the current RCL. Now, a new RCL is populated with the α nodes closest to v_r, omitting any node that has been already selected in the previous iterations. If the number of unselected nodes is fewer than α then the size of current RCL is decreased. Therefore, instead of using the fixed size of RCL, we introduce a greediness factor α_g which represents the percentage of unselected nodes that can populate RCL. We then dynamically compute the size of RCL as $\alpha = \lceil \alpha_g \cdot |V_u| \rceil$ where V_u represents a set of unselected nodes. Finally, the complete GRASP-based construction of the initial solution is detailed in the Algorithm 2.

3.3 Improvement/Search phase

In the improvement/search phase, we adapt a Variable Neighborhood Search (VNS) based approach [15, 21] to further improve the previously generated initial solution. We search the solution space with a systematic change of *neighborhood*, by both descending to local optima and escaping from valleys which contain them.

The basic scheme of VNS is given in Algorithm 3. At the initialization of the algorithm, a set of neighborhood structures N_k ($k = 1, \ldots, k_{max}$) has to be defined for the solution space S (line 1).

ALGORITHM 3: The basic VNS metaheuristic

1 Define neighborhood structures N_k ($k = 1, \ldots, k_{max}$)
2 Generate initial solution $s \in S$
3 Choose a stopping condition
4 **while** stopping condition is not met **do**
5 $k \leftarrow 1$
6 **while** $k \leq k_{max}$ **do**
7 $s' \leftarrow$ Perturbate(s); $s' \in N_k(s)$
8 $s'' \leftarrow$ LocalSearch(s'); $s'' \in S$
9 **if** *Score*(s'') > *Score*(s) **then**
10 $s \leftarrow s''$
11 $k \leftarrow 1$
12 **else**
13 $k \leftarrow k + 1$
14 **end**
15 **end**
16 **end**

In other words, for each solution, $s \in S$, $N_k(s)$ represents a set of neighbor solutions. In most implementations of VNS, successive neighborhoods N_k are nested, i.e., $N_1(s) \subset N_2(s) \subset ... \subset N_{k_{max}}(s)$. After the initialization, starting from the initial solution s and the first neighborhood structure $N_1(s)$, the algorithm systematically explores solution space S by moving within chosen neighborhood structures. At each iteration, the *Perturbate* procedure (sometimes called *Shake* procedure) randomly selects a new solution s' within the current neighborhood $N_k(s)$ (line 7). Then, a descent from s' is done by the *LocalSearch* routine which leads to a new local optimum s'' (line 8). At this point, the new solution s'' is compared with the current solution s (line 9). If s'' is better than s, the algorithm re-centers the search around s'' by replacing s with s'' (line 10), and starts all over from the first neighborhood structure $N_1(s)$ (line 11). If s'' is not better than s, the algorithm moves to the next neighborhood structure (line 13) and selects a new random solution from $N_{k+1}(s)$. The algorithm iterates until a predefined termination condition is met. An example of the termination condition could be a number of iterations until the solution has not been improved. The choice of neighborhood structures, as well as the implementation of local search and perturbation, affect the performance of VNS. Therefore, we introduce our design of VNS. We first define the local search moves employed to build solution neighborhoods.

Search moves. We use two local search moves with combining: *shift* and *2-opt*. We aim to minimize waiting cost with one of them and try to minimize travel cost with the other. First, the *shift* move generates a neighbor solution by relocating (i.e., rescheduling) a single visit within one of the solution's routes. For example, in Figure 3a, a new route p'_k is created from route p_k by shifting visit to node v_i two positions to the right (after visit to node v_{i+2}). Here, the newly created route p'_k is part of a new neighbor solution P'. Notice that we denote by (v_i, p_k) a visit to node v_i in route p_k. The *shift* move can be efficiently applied to reduce waiting times at congested nodes (i.e., loading areas). We can reschedule costly visits to less busy times by simply shifting them using the introduced move. The *2-opt* move, on the other hand, aims at reducing the travel time between selected nodes. *2-opt* was originally proposed for solving the traveling salesman problem (TSP) [8], but has been successfully applied in many other route optimization problems. The main idea behind the *2-opt* move in our solution is to reduce the total travel time by detecting and then reordering non-optimal parts of a solution where its routes cross over themselves. *2-opt* move removes two edges from a route, and reconnects the two new created sub-routes. For example, in Figure 3b, *2-opt* removes edges $e_{(i-2)(i-1)}$ and $e_{(i+1)(i+2)}$ and reconnects the route with a pair of new edges $e_{(i-2)(i+1)}$ and $e_{(i-1)(i+2)}$. Note also that *2-opt* reverses a segment of the route between the exchanged edges.

Neighborhood structures and local search. Using the two-layer search moves together, we can reduce both waiting and travel time of a solution. Therefore, we use them both to move from one solution to another in the search for local optimum. This means that the neighborhood of a solution s is composed of all solutions that can be obtained by applying one *shift* and one *2-opt* move to solution s. However, such neighborhood is far too large to be completely explored at each iteration of VNS. For example, for a

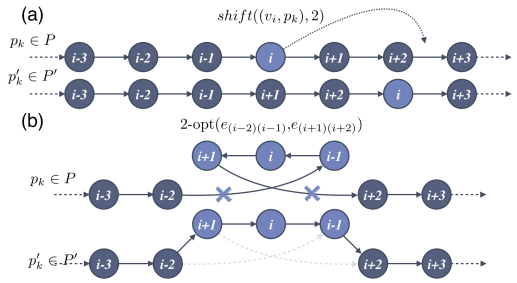

Figure 3: Local search moves: (a) *shift* and (b) *2opt*

single route with n nodes, we can reschedule (i.e., shift) visits to $n-1$ nodes at $n-2$ different positions in route s, yielding the total of $(n-1)(n-2)$ possible *shift* moves. On the other hand, we can exchange $n(n-3)/2$ different pairs of edges in the *2-opt* move. Finally, this means that by applying the search moves to only one of the solution's route, we can obtain a total of $n(n-1)(n-2)(n-3)/2$ different neighbor solutions. This is why in the local search procedure we choose to restrict the size of neighborhood in the following way. Since the goal of *shift* move is to reduce the total waiting time of a solution, we select random k visits with the longest waiting times, and consider only *shift* moves that reschedule the selected visits. Here, we again employ the concept of RCL and

ALGORITHM 4: Local search

Input: Random solution P_r, VNS depth k
Output: Improved solution P
1 **Procedure** LocalSearch(P_r,k)
2 $P \leftarrow P_r$
3 **while** P is improved **do**
4 $P_c \leftarrow$ bestShift(P,k)
5 $P_c \leftarrow$ best2Opt(P_c,k)
6 **if** $f(P_c) < f(P)$ **then**
7 $P \leftarrow P_c$
8 **end**
9 **end**
10 return P

Input: Input solution P_r, VNS depth k
Output: Improved solution P
1 **Procedure** bestShift(P_r,k)
2 $\alpha \leftarrow$ random size of RCL
3 $RCL \leftarrow \alpha$ visits with the highest waiting time
4 $RCL_k \leftarrow k$ random visits from RCL
5 $P \leftarrow P_r$
6 **foreach** $(v_i, p_k) \in RCL_k$ **do**
7 **foreach** possible shift length m **do**
8 $P_c \leftarrow$ shift(v_i,p_k,m)
9 **if** $f(P_c) < f(P)$ **then**
10 $P \leftarrow P_c$
11 **end**
12 **end**
13 **end**
14 return P

Input: Input solution P_r, VNS depth k
Output: Improved solution P
1 **Procedure** best2Opt(P_r,k)
2 $\alpha \leftarrow$ random size of RCL
3 $RCL \leftarrow \alpha$ traversed edge pairs with the highest travel cost
4 $RCL_k \leftarrow k$ random edge pairs from RCL
5 $P \leftarrow P_r$
6 **foreach** (edge pair (e_{ij}, e_{mn}), p_k) $\in RCL_k$ **do**
7 $P_c \leftarrow$ 2opt(p_k,e_{ij},e_{mn})
8 **if** $f(P_c) < f(P)$ **then**
9 $P \leftarrow P_c$
10 **end**
11 **end**
12 return P

randomly select k visits from the list of visits sorted by waiting times. From the reduced neighborhood, we then choose a solution with the lowest total time cost (i.e., objective function value). This approach is outlined in the `bestShift` procedure of Algorithm 4.

Since the goal of *2-opt* is to reduce the total travel time of a solution, we randomly select k edge pairs from RCL among the traversed edges with the highest relative travel cost. The relative cost is computed by the relative distance between the current edge cost and the cost of the shortest incoming edge to the directed node. We avoid to select the edges with the largest cost directly because for a node, all incoming edges may have large costs. In that case, the incoming edge with a large cost is less likely to indicate it is suboptimal. After selecting, we construct new solutions by applying *2-opt* moves to the selected edge pairs only in the selected routes. Again, a solution with the lowest time cost is chosen from the reduced neighborhood. The approach is outlined in the `best2Opt` procedure of Algorithm 4. Also in Algorithm 4, the local search procedure is executed until a local optimum of the current neighborhood is encountered, i.e., until current solution P cannot be improved with the employed local search moves. Finally, notice that the size of a neighborhood depends on the current depth of VNS, k. This necessarily means that employed neighborhood structures N_k are nested, which helps VNS to avoid being trapped in local optima by a systematic change of neighborhood.

Perturbation. As we previously discussed, it is important for VNS implementations to balance intensification (i.e., local search) and diversification (i.e., perturbation) during the search process to allow escaping from local optima. In our implementation of VNS, we perturbate solutions by shuffling a certain percentage of their routes, based on the current depth of VNS, k. More precisely, we randomly select $\lceil M/k \rceil$ routes to be shuffled, where M represents the total number of routes in the solution. Thereafter, the selected routes are shuffled by randomly rescheduling the visits using the *shift* move. Since our implementation of perturbation is straightforward, we omit the algorithm pseudo code to save space.

ALGORITHM 5: Overall Approach

Input: Route query set Q, maximum number of successive iterations without improvement $maxIter$, maximum number of neighborhood degree k_{max}
Output: Optimal solution P

```
1  Procedure FindSolution(Q,maxIter,k_max)
2      iter ← 0
3      best ← ConstructRandomSolution(Q)
4      while iter < maxIter do
5          iter ← iter + 1
6          P ← ConstructRandomSolution(Q)        } Construction phase (GRASP)
7          k ← 1
8          while k ≤ k_max do
9              P_r ← Perturbate(P,k)
10             P_c ← LocalSearch(P_r,k)
11             if f(P_c) < f(P) then
12                 P ← P_c
13                 k ← 1                         } Improvement phase (GRASP+VNS)
14                 iter ← 0
15             else
16                 k ← k + 1
17             end
18         end
19         if f(P) < f(best) then
20             best ← P
21         end
22     end
23     return best
```

The overall approach which combines the construction and improvement phases is outlined in Algorithm 5.

4 EXPERIMENTAL EVALUATION

We conducted extensive experiments on real-world data collected across Barcelona's urban freight transport network. We first present experiments on synthetic data to illustrate the trade-offs and parameters involved in the accuracy of the proposed solution. We then present the analysis and results on Barcelona data that provide useful insights and show practical applicability of our solution.

4.1 Experiments on Synthetic Data

Experimental Setup. We randomly generated 50 test instances, each composed of 5, 10, 15, 20 or 25 concurrent route queries (see Definition 2.2). In order to increase solving complexity, we provoked more congestions by limiting the capacity of each area (l_i) to only one parking space. While at first this dataset may seem insufficiently large, notice that the formulated test instances are NP-hard problems and as such are computationally extremely expensive. Nevertheless, we need to solve them to optimality in order to evaluate the accuracy of our approach. Therefore, we developed an integer programming model for the problem under study and solved them using IBM's commercial solver CPLEX Optimizer. The computation was limited to 6 hours and 220 GB of memory per test instance, and thus computing the optimal solutions for all 50 test instances lasted more than 12 days in total on a machine with 2.40 GHz Intel® Xeon® E5-2630 processor and 265 GB of RAM.

Finally, we assess the quality of our approach with the accuracy measure defined as follows,

$$accuracy(P, Q) = \frac{\sum_{p \in P_{opt}} cost(p)}{\sum_{p_k \in P} cost(p_k)}, \tag{6}$$

where P and P_{opt} are sets of solution routes for the route queries in Q, obtained by our algorithm and CPLEX solver, respectively.

In addition to the above comparison, we also compute the accuracy of the solutions composed of individually optimal routes with the shortest travel times. The aim of such comparison is to validate our hypothesis that, in urban transport networks with limited parking facilities, delivery routes should be planned collectively, rather than individually, and thus to efficiently balance waiting and travel cost. In the rest of this section, we refer to the individually optimal solutions as HPP, and to the solutions obtained with the proposed Multi-Vehicle Route Planner as MVRP.

Results. Figure 4 plots (a) the accuracy and (b) running time of MVRP iterations for different number of nested neighborhood structures k_{max} (see section 3.3). Figure 4a shows that the proposed algorithm is able to obtain high quality solutions (> 90% accuracy) in less than 50 iterations. It is also interesting to notice that with the smaller number of neighborhood structures (e.g., $k_{max} = 5$), MVRP is able to quickly find good solutions, but it is unable to find the optimal one in 300 iterations for particular example. This is due to very frequent change of search neighborhoods, which are sometimes left insufficiently examined. In Figure 4a, this can be observed as successive steps (i.e., jumps) in accuracy line plot. On the other hand, larger number of neighborhood structures (e.g., $k_{max} = 20$), allows MRVP deeper to explore promising search

areas and thus to find the optimal solution (in this particular case). However, as shown in Figure 4b this comes at an expense of longer computation time. For example, running 300 iterations took 748 ms with $k_{max} = 5$, while with $k_{max} = 20$ took twice as much (1,667 ms). Obviously, as we increase the maximum number of nested neighborhood structures, the iterations become longer as the algorithm needs to descend into deeper search areas. Finally, notice from Figure 4 that we are in general able to find high quality solutions in less than 1.5 seconds for synthetic test instances.

In Figure 5 we plot the accuracy of MVRP and HPP approach for test instances of different sizes (i.e., number of route queries). As shown in the figure, the accuracy of the solutions with individually optimal routes (HPP) is significantly lower than the accuracy of the solutions obtained by MVRP. For smaller test instances with 5 route queries, HPP approach yields relatively good solutions with ~ 96% accuracy. However, as we increase the number of route queries, the accuracy drops to ~ 88%. Notice here that as we increase the number of concurrent route queries, we also increase the possibility for congestions and longer waiting times at loading areas. Obviously, for larger test instances, HPP approach fails to avoid congestions and reduce waiting times. On the other hand, MVRP can find high quality solutions, even for more complex test instances with 10 and more route queries. Further, as we increase the search depth (k_{max}) and iteration limit `maxIter` (see Algorithm 5), we can obtain even better solutions, however, at an expense of longer computation time, as previously explained.

4.2 Experiments on Real-world Data

Experimental setup. We collected real-world data for *areaDUM* via the mobile parking management system that was specifically developed for freight deliverers in the city of Barcelona [22]. It is crowdsourced from citizens and deliverers that reflects the usage of the urban freight infrastructure. The data consists 3,704,034 parking check-ins by 49,172 users, i.e., deliverers, in 2,038 loading areas with the total of 8,707 parking spaces, over the period between January 1st, 2016 and July 15th, 2016. Each check-in consists of the ID of the deliverer who made the check-in, her vehicle's plate number (anonymized), the loading area to which the check-in was made, geographic coordinates, and a timestamp. Using the *areaDUM* dataset, one can discover the actual routes taken by deliverers during the course of their work. We inferred the sequences in which the loading areas were visited by grouping check-ins made in the same day by the same deliverer. However, the loading areas are often occupied by construction operators and casual deliverers who usually perform only few stops per day [18]. Therefore, we filtered out shorter routes and focused only on those with four or more stops. This way, we obtained the total of 181,454 routes. Further, in order to estimate durations of the deliveries performed at each stop, we needed to find typical travel times between each pair of loading areas in Barcelona's urban transport network. Here, we assumed that deliverers act economically and usually take the shortest path between loading areas. Therefore, we employed Open Source Routing Machine (OSRM)[2] with Barcelona's road network to compute the shortest travel times between each pair of loading areas. Thereafter, we estimated delivery durations as $d_i^k = (t_{i+1}^k - t_i^k) - c_{i,i+1}$ where t_i^k and t_{i+1}^k are arrival times at loading area v_i and v_{i+1}, respectively, and $c_{i,i+1}$ is the travel time at the shortest path between them.

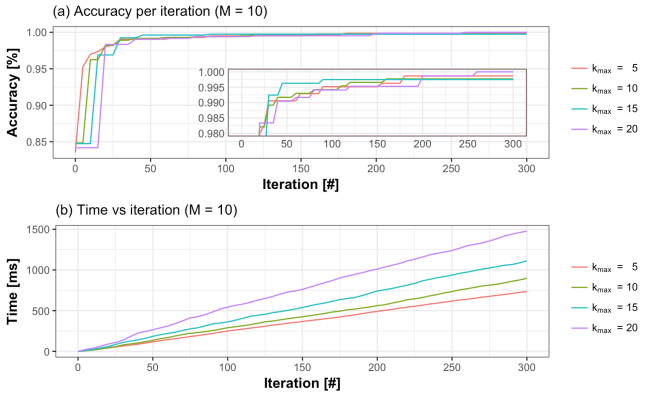

Figure 4: Accuracy improvement with respect to number of iterations (a) and computation time (b)

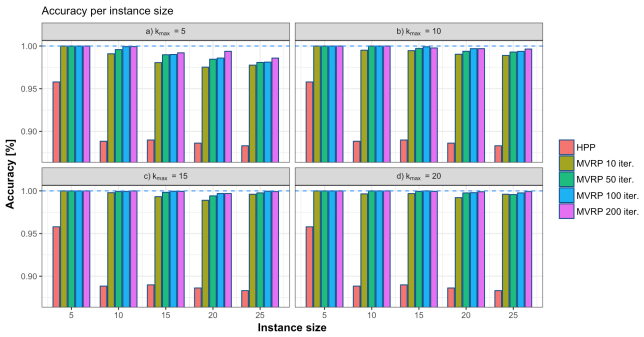

Figure 5: Accuracy improvement over whole synthetic dataset with different setups

Finally, we generated 10,000 test instances by sub-setting the discovered delivery routes. The complexity of the instances varies in terms of the number of routes, i.e., route queries, and similarities between them. 1,000 test instances were generated for 10 different problem sizes (with 10, 20, 30, ..., and 100 route queries). Obviously, in instances with highly similar route queries, i.e., with frequent mutual stops, congestions at the delivery areas occur more often and, thus, are harder to avoid. We solved all of the instances using the proposed solution, and then compared the obtained solution routes with the real-world routes traversed by actual deliverers in the city of Barcelona, as well as with the individually optimal routes computed with CPLEX using the HPP formulation.

[2]http://project-osrm.org/

Table 1: Route improvement comparison

size	REAL vs		HPP vs		MVRP vs	
	HPP	MVRP	REAL	MVRP	REAL	HPP
10	4.1%	4.1%	95.9%	0.4%	**95.9%**	**7.1%**
20	1%	0%	99%	4.1%	**100%**	**60.4%**
30	0.7%	0%	99.3%	13%	**100%**	**66.7%**
40	3.5%	0%	96.5%	15.9%	**100%**	**83.5%**
50	1%	0%	99%	9.8%	**100%**	**90.2%**
60	5.4%	0%	94.6%	9%	**100%**	**91%**
70	0.7%	0%	99.3%	15.4%	**100%**	**84.6%**
80	1.1%	0%	98.9%	4.3%	**100%**	**95.7%**
90	2.1%	0%	97.9%	16.9%	**100%**	**83.1%**
100	3.91%	0%	96.09%	10.03%	**100%**	**89.97%**

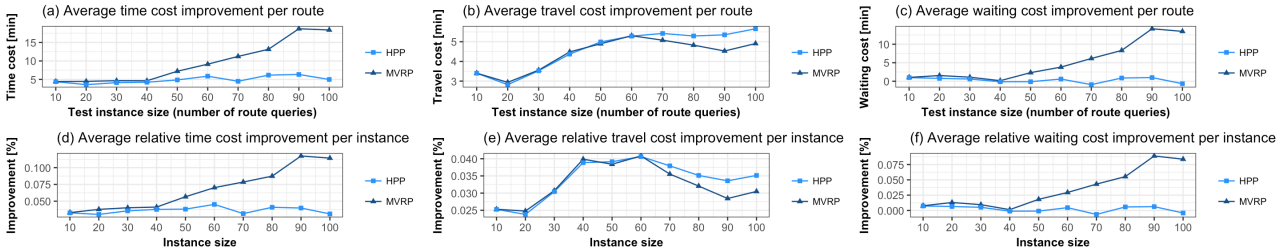

Figure 6: Route improvement in terms of average (a-c) and relative (d-f) total, travel, and waiting costs.

Results. Table 1 summarizes the pairwise comparison between solution routes obtained by our algorithm (MVRP), actual routes derived from the *areaDUM* dataset (REAL), and individually optimal routes (HPP). The comparison was made in terms of the percentage of test cases in which one route set is more efficient (i.e., less time consuming) than another. For instance, for the smallest test instances (10 route queries), MVRP routes are better than HPP in 7.1% of the cases, while HPP routes are more efficient in only 0.4% of the cases. In the rest of the cases, MVRP and HPP routes were equally efficient.

In general, both HPP and MVRP routes were almost always less time consuming than REAL routes, regardless of the instance size, i.e., number of route queries. However, in rare occasions, REAL routes are still better than HPP routes. This was expected, since in HPP we assume that deliverers act opportunistically and aim at minimizing the total travel time, which can occasionally lead to congestions and waiting times at loading areas. On the other hand, REAL routes were more efficient than MVRP routes in only 4.1% of the times for the smallest test instance with 10 route queries. It is also worthwhile to note that MVRP routes were in most cases more or equally efficient than HPP routes. This can be clearly observed by comparing the percentages in columns HPP-vs-MVRP and MVRP-vs-HPP. However, such similarity in performance is only observable for smaller test instances with fewer route queries, where there is a low probability for congestions and waitings at loading areas. In these cases, the most efficient route arrangement is composed of individually optimal (i.e., shortest in length) routes. For larger test instances with more than 30 route queries, MVRP routes are in 90% of test cases more efficient than REAL and HPP routes.

While both arrangements improve over actual routes taken, we want to observe the extent to which route arrangement provides superior improvements. We plot the average time cost improvements compared to actual route arrangements inferred from the *areaDUM* dataset (REAL routes) in Figure 6. In Figure 6a and d, we first plot the average overall time cost reduction per route for different instance sizes in minutes and percentages, respectively. At the beginning of the curve, HPP and MVRP perform similarly for smaller test instances with up to 30 route queries. In such cases, the two methods can reduce the total durations of real-world routes for up to 4% (5 minutes) in average. However, for larger test instances, the difference between the performance of HPP and MVRP becomes more observable. This can be seen in the plot as a separation of the curves for test instances with more than 40 route queries. For example, for the largest test instances with 100 route queries, MVRP reduced the average time cost per route by $\sim$ 11.5% (18.4 minutes) while HPP performed at the same level as for smaller instances reduced the cost only for close to 3 % (4 minutes). Such behavior can be explained by the fact that most of the loading areas in Barcelona contain around 4 parking spaces, and as such, they are usually large enough to accommodate vehicles scheduled in smaller test instances. On the other hand, congestions and waitings at loading areas are more common in larger test instances, and thus the HPP approach, which aims only at minimizing the travel costs, performed significantly worse than MVRP.

Figure 6b-c and e-f separate travel and waiting cost improvements to further investigate the performance of HPP and MVRP. Figure 6b and c suggest that both methods performed comparably in terms of travel time reduction for test instances with up to 60 route queries. For larger test instances this reduction is less significant for MVRP, since it aims to rearrange routes to minimize waiting times. This can be clearly observed in Figure 6c and d. In the same plot, we see that HPP is unable to reduce waiting costs, regardless of the instance size, which was expected. The two plots show that MVRP is able to strike a balance between travel and waiting costs, which directly confirms the effectiveness of the employed *shift* and *2-opt* search moves. Furthermore, the effectiveness of the proposed solution grows with the number of concurrent route queries, which makes it especially suitable for routing applications in dense urban environments.

5 RELATED WORK

To the best of our knowledge, the new multi-vehicle routing problem and its urban freight delivery application have not been addressed before both in research and practice. On the theoretical side, a minimum-weight Hamiltonian path problem is used to compute individually optimal (i.e., shortest in length) delivery routes, by assuming that deliverers act economically and try to minimize the total travel time. Traveling salesman problem (TSP) is a special case of Hamiltonian cycle problem defined on edge-weighted graphs. A variation of TSP, where the initial point does not have to be revisited at the end of the cycle, is known as Traveling Salesman Path Problem (TSPP). A related combinatorial graph problem is multiple TSP (mTSP) [3]. In mTSP, a set of routes needs to be determined as to visit all of the planned nodes and minimize the total cost. Here, each node can be visited exactly once in only one route, while each route needs to start and end at the same depot. The problem is composed of both dividing the set of planned visits among multiple travelers and solving TSP for each of them. Even though local search methods [4] can be employed to solve mTSP, various metaheuristics such as genetic algorithms (GA) [25] and ant colony optimization [19] have been proposed. The proposed problem assigns each route with its own set of nodes that need to

be visited. We also introduce the time dependencies between the routes by assigning capacities constraints to nodes, which, to the best of our knowledge, have not been considered in the literature.

In a special case of TSP with time windows (TSPTW) [9], each node needs to be visited within the given time interval $[a, b]$. Although arriving after b is not acceptable, a vehicle can arrive to a node before a but it needs to wait until a. Kara et. al. formalize the problem of minimizing the total tour cost including the waiting times [16]. Here, the arrival to the next node is computed by summing the travel costs of previously traversed edges and and waiting times. We also include delivery durations which provides a more realistic model for the problem.

An VNS method that employs Greedy Randomized Adaptive Search Procedure (GRASP) was proposed for Traveling Repairman Problem (TRP), to build initial solutions and thus speed up the search convergence [24]. The GRASP's concept of restricted candidate list is utilized in our approach to combine randomization and greediness when selecting the promising search moves in the local search phase. To build solution neighborhoods at each iteration, we use two local search moves: *shift* and *2-opt*. *2-opt* was originally proposed for TSP [8], but has been successfully in other route optimization problems. We used *2-opt* to minimize the travel time of routes. On the other hand, the *shift* move was introduced to reschedule visits to busy nodes, and thus to avoid congestions and reduce the waiting times.

All of the beforementioned problems are NP-complete, and thus different heuristic algorithms have been proposed in the literature. They can be categorized into three groups: tour construction (e.g., nearest-neighborhood, convex-hull), tour improvement (e.g., k-opt, Or-opt), and metaheuristic-based algorithms (e.g., simulated annealing [20], variable neighborhood search (VNS) [6, 11, 24], tabu search [5, 13], genetic algorithms [2, 14, 14]). We develop a greedy randomized constructive approach to build an initial solution before the actual search process, and improve the search process by positioning the search in the promising solution space areas. Also our approach for selecting the right node as well as the shift position have not been addressed in research nor practice before.

6 CONCLUSION

A new problem formulation and practical solution are presented to solve the congestion problem in real-world urban freight transport networks. The analysis and experiments on data collected across Barcelona's urban freight transport network confirmed their effectiveness and ability to significantly reduce the average length of delivery routes. For example, for moderately-sized problems with only 100 concurrent delivery routes, the proposed system saves up to 19 minutes per route, which sums up to 1,900 minutes of idling[3] time across all 100 routes. According to [17], 1,900 minutes of heavy-duty vehicle idling can produce 146.8 kg of CO_2 which can be a significant contribution to overall emission reduction. The developed solution achieves a more effective urban freight mobility for 90% of cases compared to both the individually optimal and real life traces of delivery. The improvement achieved by the proposed solution grows with the number of concurrent route queries, which makes it especially suitable in dense urban environments.

[3]Idling refers to running a vehicle's engine when the vehicle is not in motion.

Our solution opens a new exciting direction of collective planning of traffic and urban freight transport, and makes the specific case of improving transportation with a global optimization task and fairness to all vehicles in the system. For future work, we plan to develop an online version of the proposed approach which will consider the real-time traffic information crowdsourced from deliverers or collected from employed sensors. We also plan to make it publicly available for use in other cities.

REFERENCES

[1] European Environment Agency. 2016. Freight transport demand. (2016). https://www.eea.europa.eu/data-and-maps/indicators/freight-transport-demand-version-2/assessment-6

[2] Zakir H Ahmed. 2010. Genetic algorithm for the traveling salesman problem using sequential constructive crossover operator. *International Journal of Biometrics & Bioinformatics (IJBB)* 3, 6 (2010), 96.

[3] Tolga Bektas. 2006. The multiple traveling salesman problem: an overview of formulations and solution procedures. *Omega* 34, 3 (2006), 209–219.

[4] Leonora Bianchi, Joshua Knowles, and Neill Bowler. 2 005. Local search for the probabilistic traveling salesman problem: Correction to the 2-p-opt and 1-shift algorithms. *European Journal of Operational Research* 162, 1 (2 005), 206–219.

[5] José Brandão. 2004. A tabu search algorithm for the open vehicle routing problem. *European Journal of Operational Research* 157, 3 (2004), 552–564.

[6] Francesco Carrabs, Jean-François Cordeau, and Gilbert Laporte. 2007. Variable neighborhood search for the pickup and delivery traveling salesman problem with LIFO loading. *INFORMS Journal on Computing* 19, 4 (2007), 618–632.

[7] CIVITAS Initiative WIKI consortium. 2015. *Smart choices for cities: Making urban freight logistics more sustainable.*

[8] Georges A Croes. 1958. A method for solving traveling-salesman problems. *Operations research* 6, 6 (1958), 791–812.

[9] Yvan Dumas, Jacques Desrosiers, Eric Gelinas, and Marius M Solomon. 1995. An optimal algorithm for the traveling salesman problem with time windows. *Operations research* 43, 2 (1995), 367–371.

[10] Thomas A Feo and Mauricio GC Resende. 1995. Greedy randomized adaptive search procedures. *Journal of global optimization* 6, 2 (1995), 109–133.

[11] Krzysztof Fleszar, Ibrahim H Osman, and Khalil S Hindi. 2009. A variable neighbourhood search algorithm for the open vehicle routing problem. *European Journal of Operational Research* 195, 3 (2009), 803–809.

[12] Michael R Garey, David S. Johnson, and Larry Stockmeyer. 1976. Some simplified NP-complete graph problems. *Theoretical computer science* 1, 3 (1976), 237–267.

[13] Michel Gendreau, Gilbert Laporte, and Frédéric Semet. 1998. A tabu search heuristic for the undirected selective travelling salesman problem. *European Journal of Operational Research* 106, 2-3 (1998), 539–545.

[14] John Grefenstette, Rajeev Gopal, Brian Rosmaita, and Dirk Van Gucht. 1985. Genetic algorithms for the traveling salesman problem. In *Proceedings of the first International Conference on Genetic Algorithms and their Applications*. 160–165.

[15] Pierre Hansen, Nenad Mladenović, and José A Moreno Pérez. 2010. Variable neighbourhood search: methods and applications. *Annals of Operations Research* 175, 1 (2010), 367–407.

[16] Imdat Kara and Tusan Derya. 2015. Formulations for Minimizing tour duration of the traveling salesman problem with time Windows. *Procedia Economics and Finance* 26 (2015), 1026–1034.

[17] ABM S Khan et. al. 2006. Idle emissions from heavy-duty diesel vehicles: review and recent data. *Journal of the Air & Waste Management Association* 56, 10 (2006), 1404–1419.

[18] Burcu Kolbay et. al. 2017. Analyzing Last Mile Delivery Operations in Barcelona's Urban Freight Transport Network. In *2017 EAI 2nd International Conference on ICT Infrastructures and Services for Smart Cities*. EAI.

[19] Weimin Liu, Sujian Li, Fanggeng Zhao, and Aiyun Zheng. 2009. An ant colony optimization algorithm for the multiple traveling salesmen problem. In *4th IEEE Conference on Industrial Electronics and Applications*. IEEE, 1533–1537.

[20] Miroslaw Malek, Mohan Guruswamy, Mihir Pandya, and Howard Owens. 1989. Serial and parallel simulated annealing and tabu search algorithms for the traveling salesman problem. *Annals of Operations Research* 21, 1 (1989), 59–84.

[21] Nenad Mladenović and Pierre Hansen. 1997. Variable neighborhood search. *Computers & operations research* 24, 11 (1997), 1097–1100.

[22] Petar Mrazovic, Bahaeddin Eravci, Hakan Ferhatosmanoglu, Josep Lluis Larriba-Pey, and Mihhail Matksin. 2017. Understanding and Predicting Trends in Urban Freight Transport. In *2017 IEEE 18th International Conference on Mobile Data Management*. IEEE.

[23] Mauricio GC Resende and Celso C Ribeiro. 2014. GRASP: Greedy randomized adaptive search procedures. In *Search methodologies*. Springer, 287–312.

[24] Amir Salehipour, Kenneth Sörensen, Peter Goos, and Olli Bräysy. 2011. Efficient GRASP+VND and GRASP+VNS metaheuristics for the traveling repairman problem. *4OR: A Quarterly Journal of Operations Research* 9, 2 (2011), 189–209.

[25] Mohammad Sedighpour, Majid Yousefikhoshbakht, and Narges Mahmoodi Darani. 2012. An effective genetic algorithm for solving the multiple traveling salesman problem. *Journal of Optimization in Industrial Engineering* 8 (2012), 73–79.