Dynamic Delivery Services for Smart Cities

Solving a Constrained Pickup Delivery Vehicle Routing Problem with Time Windows on a Dynamic Environment

Miguel de Sousa Esteves Martins
miguelsemartins@tecnico.ulisboa.pt
Instituto Superior Técnico, Universidade de Lisboa, Lisboa, Portugal
November 2017

***Abstract*—In this thesis, it is proposed a model for courier services capable of handling the routing of a fleet of vehicles. More specifically, a Capacitated Pickup Delivery Vehicle Routing Problem with Time Windows on a Dynamic Environment is formulated. This problem emerged from a real-world problem existent in the company *Urban Dynamic Delivery*. This specific case study requires a limited, short time for solving the optimization problem. Thus, a good solution must be found in a short time window, instead of finding the optimal solution independently of the required computational time. A new method pairing nearest neighborhood search with subtractive clustering is proposed to improve initial solutions and to accelerate the convergence of the optimization algorithm. Further, a hybrid method combining Ant Colony Optimization with Local Search is proposed to model this problem. First, a model was developed to solve a given static instance. Then, a framework to coordinate any dynamic changes on the inputs over time is proposed. A minimal functional model for the tackled problem was derived and is applied as it is to a case study. The proposed clustering approach improve the results. Further, the proposed optimization algorithm presents good results for the dynamic environment and is suitable to be applied to the real-world scenario.**

***Index Terms*—pickup delivery problem, ant colony optimization, local search, time windows, dynamic requests**

I. Introduction

The objective of this thesis is to develop a model for courier services based on the general guidelines provided by the company *Urban Dynamic Delivery*. It deals with transporting goods between two distinct locations while efficiently handling the routing of a vehicle fleet. It is constructed to handle both changing traffic conditions and insertion of new customer requests in realtime.

A new model was created to model a routing problem for handling pickups and deliveries. It is formulated to respect most common constraints required for practical purposes, such as obeying target delivery times and having limited load capacity. Ultimately, our focus will be to consistently get a good solution under a tight time interval instead of achieving the optimal result, which could take a long period of time. For validation, the developed strategy is tested against publicly available benchmarks of 100 customers for static problems [1]. Besides this final value comparison, our generated routes are also validated as feasible according to a program directly granted from the entity behind the page providing the benchmarks. Concepts were implemented and tested in MATLAB. Hierarchical goal ordering is: minimizing lateness to serve all current requests; number of used vehicles; total travelled distance. The weights given to each objective are an input to the model for greater flexibility.

Besides defining a new mathematical formulation applicable to the modelled problem, it is showed that pairing the ant colony metaheuristic with local neighbourhood search is a valid approach. Also, a novelty concept for initial solution construction is proposed based on nearest neighbourhood search and subtractive clustering

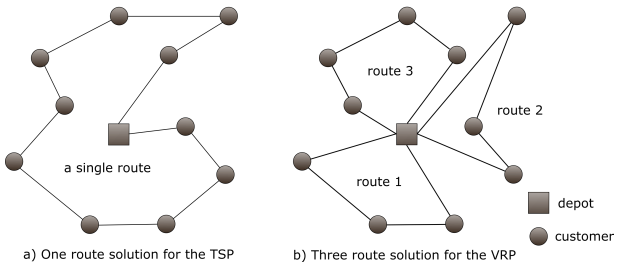
II. Background

A. Vehicle Routing Problem

The Vehicle Routing Problem (VRP) category is a combinatorial optimization and integer programming set of problems. It deals on how to direct a fleet to serve interest points in the most profitable way. It has many uses in industry, for obvious reasons. The main hurdle when solving a combinatorial problem is the sheer number of solutions that compose the feasible solution set, from where the ultimate goal is to extract the optimal solution. Many of these problems are in fact NP-Hard, i.e., there is no guarantee the optimal result can be reached in polynomial computation time [2].

The first time this theme has approached was when dealing with the Travelling Salesman Problem (TSP), in [3], where the goal is to make a single salesman find the round-trip through each and all the cities only once. From this initial concept a broader generalization was made in 1959, the Vehicle Routing Problem (VRP), credited to [4]. The focus of the VRP shifts to finding the optimal set of routes for a fleet of vehicles to service

Fig. 1. TSP and VRP example for the same set of points



a given set of customers. The new concept expands the TSP by introducing the notion of *delivery*. We are no longer interested only on the path between each city and now need to take into account what needs to be done at each location. An example comparison between the two problems can be visualized in Figure 1 for the same set of points.

From this starting point, over the years extra constraints were added to represent and solve diverse scenarios. Many of the additional constraints to add to VRP can be expanded into their own categories [5] [6]:

- **Capacitated VRP (CVRP)**: each customer has a specific demand for an amount of goods, which are characterized by a certain volume or weight. All goods originate from a defined location called the *depot*. Vehicles have finite capacity, meaning they can only take a certain volume and/or weight at a time.
- **Time Windows**: when each interest point needs to be visited between a specified time interval. A distinction can be made for *hard time windows*, where it is mandatory that each node is serviced within the defined time window limits, or *soft time windows*, where interest points can be visited outside the designated time window, incurring into a penalty.
- **VRP with Pickup and Delivery (PDVRP)**: when there is a need to *pickup* an order from a specific location, besides the depot, and *deliver* it to another. A *request* is then characterized by the pair pickup-delivery. A precedence constraint must be added to the model assuring that for each pair, the pickup is visited before the delivery. Multiple requests can be serviced at once.
- **Heterogeneous fleet**: for problem formulations where the vehicle fleet specifications are not equal, such as maximum capacity, travel speed, operation costs, etc.

When all input data is known beforehand, the routing problem is said to be *static* and it means no changes will happen in the inputs, namely requests, number of vehicles, travel costs, etc. When input data is revealed or modified during a routing problem's working span, we are before a *dynamic* problem and we can no longer approach it in the same way. Usually, the input only fully revealed over time is the user requests.

To solve a dynamic problem we can adapt a method able to solve static instances into one that handles the input changes over time. The most basic way is to run the static solver every time the input conditions change (i.e., there is a new request or a cancellation). A troubling drawback from this path is that doing a complete reoptimization every time there is an update on the input conditions may not leave enough time for the static algorithm to arrive at a satisfactory solution between restarts.

An interesting approach is presented by [12], where the working span is separated into successive time intervals of fixed length and buffering all new requests for later insertion. Starting from the solution of the static algorithm for all the pre-defined requests, each time interval will receive the selected best solutions from the instance before, insert any requests that were buffered and deploy the best solution. During the rest of the interval a simulation is made to predict the system state at the end of the interval, and that prediction is then optimized as if it were a static solution. When the end of the interval is reached, the deployed solution and the solution from the reoptimization are used as input solutions of the next interval.

B. Methods to solve VRP

Computational methods used to tackle such problems can usually fit in two categories, exact algorithms and approximation algorithms [8]. With an exact algorithm it is possible to guarantee the optimal solution will be computed but in the worst case scenario, we might have to list all possible solutions before reaching the optimal one, and problems with bigger size become infeasible. Approximation methods work on a balance between quality of solution (closeness to optimality) and computational effort. These approximation methods can be grouped into three different definitions: construction techniques, local search techniques, and metaheuristics techniques [9].

- **Construction methods**: Being the most simple of the three types, these methods usually start from scratch, with the initial set either empty or starting at a random node. To compute a feasible solution, heuristic rules iteratively add new elements until all elements have been selected.
- **Local Search methods**: starting from a feasible solution, systematically explores the neighbourhood looking for improvements in the current solution. If any improvement is found, the search is continued

from the new found best solution and repeats the problem until a full search of the neighbourhood finds no improvements. Without implementing any extra strategies they are not able to escape a local minimum.

- **Metaheuristics methods**: metaheuristics make use of stochastic components in their searching process and the algorithm might temporarily choose solutions which do not improve current solution to escape local minima. These methods are able to find solutions very close to the real optimal solution but are slow when dealing with large instances.

One way to overcome the methods limitations can be to combine them. Merging the exploration of a metaheuristic with the exploitation of a local search, to complement the shortcomings of each model. This approach is coined as *hybrid method* and is widely applied to different problems [13] [14] [8]. While the metaheuristic generates new solutions, the local search ensures there is no better solution on the neighbourhood. This approach allows to efficiently solve combinatorial problems in reasonable time.

III. Implementation

A. Problem Formulation

An adaptation from the well-known mathematical formulation of the VRPTW [10] is presented, where the goal is to service as efficiently as possible a set of customers requests $\mathcal{R} = \{1, ..., n\}$.

Every request is defined by a pickup and a delivery location, each represented by a unique graph node out of a total $2n$ nodes. The full set of customers to service is given by $\mathcal{C} = \{p_1, d_1, p_2, d_2, ..., p_n, d_n\}$ where p_r is the pickup node of request r from the subset $\mathcal{P} = \{1, 3..., 2n-1\} \subset \mathcal{C}$ and d_r is the delivery node of request r from subset $\mathcal{D} = \{2, 4..., 2n\} \subset \mathcal{C}$.

In order to service each customer request we have available k vehicles. The depot location is split into k nodes forming the set of depot nodes $\mathcal{W} = \{1, ..., k\}$.

The mathematical formulation is dimensioned for a graph $\mathcal{G}(\mathcal{N}, \mathcal{A})$, where $\mathcal{A} \subseteq \mathcal{N} \times \mathcal{N}$ is the set of graph edges representing all travel possibilities between nodes and $\mathcal{N} = \mathcal{W} \cup \mathcal{C}$ represent the graph nodes. $\mathcal{V} = \{1, ..., k\}$ is the set of homogeneous vehicles, each with e_k vehicle costs and capacity $q \geq l_i,\ i \in \{1, ..., n\}$ where l_i is the load capacity demand for customer i, i.e. how much of a vehicle's available capacity a request will occupy.

The variable x_{ij}^k is a binary parameter that expresses if a vehicle travels directly from node i to node j. For each arc ij, it takes the value 1 if vehicle k travels directly from i to j and 0 otherwise. x_{kj}^k represents an arc between a depot node and node j, serviced by vehicle k. Similarly, x_{ik}^k expresses an arc between a customer node i and the depot, also serviced by vehicle k Each arc is also defined in terms of travel time, t_{ij} specific positive travel cost, c_{ij}, for each arc in $\mathcal{A}$. Finally, s_i^k is the exact time of service at each point i by vehicle k and $[a_i, b_i]$ is the time window specified for node i. The variable $weights = [\mathcal{M}_1, \mathcal{M}_2, \mathcal{M}_3, \mathcal{M}_4]$ is a vector composed by all scaling factors, which define the priority of each term in the objective function.

$$\begin{aligned}\text{minimize} \quad & \mathcal{M}_1 \times \sum_{k\in\mathcal{V}} \sum_{(i,j)\in\mathcal{A}} c_{ij} x_{ij}^k + \\ & \mathcal{M}_2 \times \sum_{k\in\mathcal{V}} \sum_{j\in\mathcal{C}} e_k x_{kj}^k + \\ & \mathcal{M}_3 \times \sum_{k\in\mathcal{V}} \sum_{(i,j)\in\mathcal{A}} \max(s_j^k - b_j, 0) x_{ij}^k + \\ & \mathcal{M}_4 \times \sum_{k\in\mathcal{V}} \sum_{(i,j)\in\mathcal{A}} \max(a_j - s_j^k, 0) x_{ij}^k \end{aligned} \tag{1}$$

$$\text{subject to} \quad \sum_{k\in\mathcal{V}} \sum_{j\in\mathcal{C}} x_{ij}^k = 1,\ \forall\ i \in \mathcal{C} \tag{2}$$

$$\sum_{(i,j)\in\mathcal{A}} l_i x_{ij}^k \leq q,\ \forall\ k \in \mathcal{V} \tag{3}$$

$$\sum_{p\in\mathcal{P}} x_{hp_n}^k - \sum_{d\in\mathcal{D}} x_{gd_n}^k = 0, \quad \forall\ h \in \mathcal{N},\ \forall\ g \in \mathcal{N},\ \forall\ k \in \mathcal{V},\ \forall\ n \in \mathcal{R} \tag{4}$$

$$\sum_{j\in\mathcal{C}} x_{kj}^k = 1,\ \forall\ k \in \mathcal{V} \tag{5}$$

$$\sum_{i\in\mathcal{V}} x_{ih}^k - \sum_{j\in\mathcal{N}} x_{hj}^k = 0,\ \forall\ h \in \mathcal{C},\ \forall\ k \in \mathcal{V} \tag{6}$$

$$\sum_{i\in\mathcal{V}} x_{ik}^k = 1,\ \forall\ k \in \mathcal{V} \tag{7}$$

$$x_{ij}^k (s_i^k + t_{i,j} - s_j^k) \leq 0,\ \forall\ (i,j) \in \mathcal{A},\ \forall\ k \in \mathcal{V} \tag{8}$$

$$a_i \leq s_i^k,\ \forall\ i \in \mathcal{N},\ \forall\ k \in \mathcal{V} \tag{9}$$

$$x_{ij}^k \in \{0, 1\},\ \forall\ (i,j) \in \mathcal{A},\ \forall\ k \in \mathcal{V} \tag{10}$$

$$x_{ij}^k \in \{0, 1\},\ \forall\ (i,j) \in \mathcal{A},\ \forall\ k \in \mathcal{V} \tag{11}$$

Typically, VRPs focus on minimizing the sum of travel costs over of the arcs used in the solution ($\sum_{k\in\mathcal{V}} \sum_{(i,j)\in\mathcal{A}} c_{ij} x_{ij}^k$). Since we are dealing with an heterogeneous fleet, we must also consider each vehicles' respective cost ($\sum_{k\in\mathcal{V}} \sum_{j\in\mathcal{C}} e_k x_{kj}^k$). The first edge of a non-empty tour is used (x_{kj}^k) in order to only add vehicle costs on relevant tours. In this proposed approach we are dealing with soft ending time windows, which means servicing a node after its time window end (b_i) does not make a solution infeasible, only unappealing. When a node is visited after its time window ends, b_i, the extra time it took between the

limit and the service time is called lateness. If it is allowed, its respective minimization must also a part of the objective function ($max(s_j^k - b_j, 0)x_{ij}^k$). The early limit of a time window, a_i, is still hard and inviolable. Finally, it might prove beneficial to specify not wanting drivers to be idle. The minimization of the total waiting times ($max(a_j - s_j^k, 0)x_{ij}^k$) appears as the last term and is especially relevant when dealing with heavily clustered data. The constraint represented in Equation 2 assures that all customers are serviced only once. Vehicle capacity constraint is represented by Equation 3. Equation 5 assures the same vehicle services both pickup and delivery nodes of the respective request. The expression 6 defines that there is only one tour per vehicle, which can be empty. Equation at 7 ensures that if a vehicle arrives at a costumer location it also departs from the mentioned customer location. Equation 8 define all tours' ending location as the depot. For a vehicle to travel directly from i to j Equation 9 states the arrival time at customer j is such that it allows travelling between i and j. Expression 10 specifies that the early limit of a time window a_i is hard and 11 denotes the x_{ij}^k variable as binary. Thus, the model is non-linear due to the non-linear *max* operations in 1 quadratic terms in 9 and integrality constraints at 11.

B. Proposed Approach

The key of the approach is the *static solver*, a module that receives an existing solution and tries to optimize it further. It runs in loop until a stopping criteria is meet and uses a hybrid Ant Colony System paired with Local Search. The *static solver* only needs to be accompanied by the *initial solution constructor* to solve static instances. Our approach to solve dynamic instances is based on the presented solution for static problems. After creating an initial feasible solution, the *static solver* is applied for a limited period of time. This intends to represent the pre-computation of requests already known beforehand. At this point we are at the beginning of the working span and will next repeat the same set of directives until all requests have been serviced:

- Insert any new requests from the previous interval into the best selected routes.
- Deploy the best obtained solution from the insertion to the physical vehicles.
- Predict current interval's end-state, namely vehicle positions and serviced requests.
- While current interval's end isn't reached, optimize the end-state prediction with the *static solver*.
- Output from the *static solver* the best found solution and the latest found solution.
- Group these two solutions with the state of the deployed route to form the best selected routes.
- Update vehicle positions, serviced requests and distance matrix.

C. Initial Solution

The *initial solution constructor* is used to generate a feasible solution from scratch, as a starting point for the *static solver* module. A proposed new strategy based on Nearest Neighbourhood Search (NNS) and subtractive clustering is presented. The better the initial solutions, the faster the convergence of the model to a good solution. With this in mind, a *clustering* of the customers nodes location is done. To each of these newly generated clusters we assign a single vehicle and guide it through its respective clustered nodes, following the NNS heuristic. A comparison between using only NNS or clustered NNS is showed in Table I, which supports the decision to use pre-clustering.

Since we are not considering the time availability of the nodes, and are clustering instead based on the location, lateness will be present on the solution. To avoid assigning too many nodes into a single vehicle to service we make an effort to go in the opposite direction and assign as many vehicles as possible to an initial solution. Starting with a very low *cluster influence range* parameter, subtractive clustering is applied on the midpoint of each pickup-delivery pair, taking into account only their location. Having an extremely low *cluster influence range* results in each midpoint being a cluster centre, i.e, it asks for as many vehicles as there are pickup-delivery pairs.

D. Local Search

Careful parameters tuning must be made to achieve a good balance between an exhaustive neighbourhood search space and computational speed when applying Local Search.

At the start, a part of the route (from now on called *slice*) of size s is removed from the original route. This slice will tentatively be inserted between every route node up to L times around the slice's original position, without leaving the current vehicle. If this slice is independent of the other customers in the same vehicle, this is, if only a pickup and the corresponding delivery are selected, the slice is also inserted on equivalent indexes at all other vehicles. The new insertion indexes are generated by a simple linear interpolation between available insertion indexes at each vehicle. When a feasible solution is generated, the corresponding fitness value is saved for later comparison.

This process is repeated for each possible slice of length s. Only after all options have been computed do we compare the best achieved result with the disturbed solution. If at this point any new feasible solution is an improvement comparing with the current disturbed one, it replaces the disturbed route and repeats the previous steps all over again. When no solution is found, slice

size is decreased and the cycle is repeated. All this is stopped only when the slice's length is decreased to 0.

E. Static Solver

The *static solver* is the key piece to the employed solution, combining both ACO metaheuristic with local search, as seen on Figure 3. Starting from a feasible solution, module specific variables are initialized. The *pheromone trail matrix* is also here generated. It is a $Q \times Q$ matrix, where Q is the length of $\mathcal{N}$, and it is initialized uniformly at the value τ_0. Pheromone limits are dynamic and depend on the quality of the current best solution, meaning they will be updated every time a new global best solution, s_{best}, is found. Pheromone trail matrix limits are computed according to [8].

First, m new ants are generated using the *new ant* generator module. For a specified limit, *LS_limit*, the module runs purely as ACO algorithm since *local search* method is not yet used. The best ant from each iteration is compared with the best overall solution. If an improving solution is found, both s_{best} and pheromone matrix limits are updated.

After all ants are computed, the *local search* method is applied to the most promising ants on the top ant fitness list from current iteration (size specified by *top_ants_number*) until no improvement is found, and are compared with the global best ant, s_{best}. From now on, any time a comparison is made between any solution and the global best, s_{best}, it is implied that if the new solutions is better it will replace s_{best} and update the pheromone limits accordingly.

If no best solution was found so far, a disturbance is induced on s_{best} ant to the output s_{new} the local search method is applied. The number of new solutions generated by the disturbance method is given by *perturbed_ants_number* times. Every s_{new} is saved on *exchange_memory* to avoid applying a local search on equal solutions and save valuable computational time. The disturbance introduced can either be a *shift* or a *switch*, both represented at Figure 2. On the first one, a randomly selected pickup-delivery pair is removed from a vehicle and attributed to another at a random location. On the second disturbance method a shift is applied two times on the same pair of vehicles, removing and inserting a new pickup-delivery pair in each. A disturbance can happen to a route multiple times to further increase the search space, as is the case of the proposed model where it can happen from one to three times.

If s_{best} is updated on current iteration, the *LS_counter* is reinitialized, *exchange_memory* cleared and q_0 equalled to its original value. If on the contrary s_{best} remains the same as in the previous operation *LS_counter* is incremented. After local search is turned

Fig. 2. Representation of disturbance method *shift* and *switch*

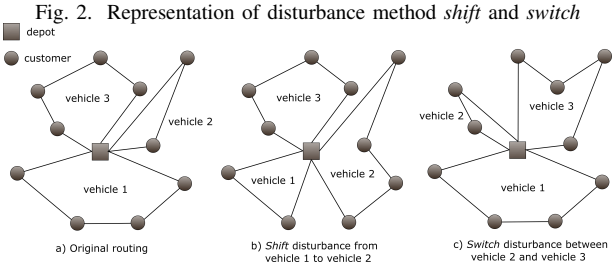
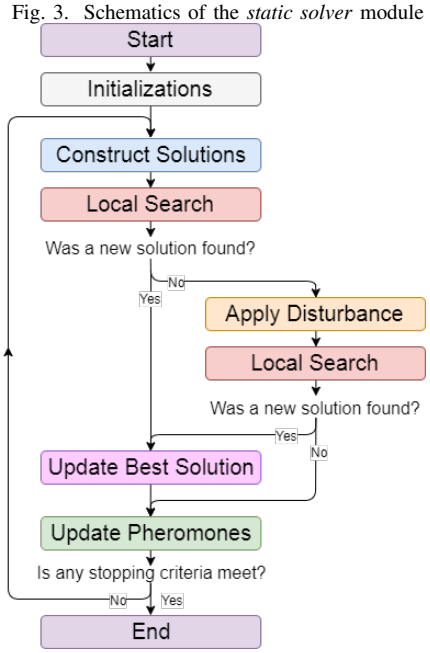


Fig. 3. Schematics of the *static solver* module



on, *LS_counter* starts to dictate when *pheromone reinitialization* will be applied. When this is not the case, the pheromone trail matrix is updated. When it is and we are before a pheromone matrix reinitialization, the pheromone matrix is equalled to the midpoint between τ_{max} and τ_{min}, the *LS_limit* is doubled and L in incremented by one and q_0 is again reinitialized. All these changes aim to increase the neighbourhood search area and the time spent searching on it before reinitialization.

When eventually the time limit is surpassed for the *static solver* module, it stops running iteratively and outputs the best found solution as result.

F. Dynamic Solver

To handle the dynamic changes over time, the working horizon is divided into successive intervals of length $time_{ss}$. During each interval, the working conditions (time and distances matrix) will remain unchanged and any new requests appearing during this interval will be buffered, i.e., saved for later insertion. When the end of the interval is reached, time and distance matrix are

updated and new requests inserted in the already existing routes. This way, during the length of the interval, the problem instance does not change and the *static solver* method can be applied.

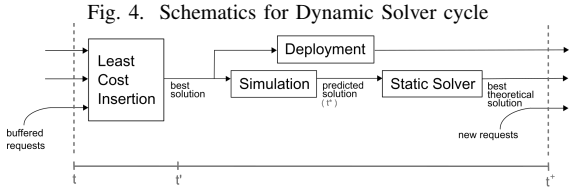
The only place where a solution is generated from scratch is at the start of the dynamic solver. Before beginning this cyclic approach described above, depending on the instance to solve, it can be relevant to run the *static solver*, depending on the number of requests or other instance related properties. This step is optional and belongs to the initializations.

After this pre-optimization time interval is over, if any occurred, we enter the iterative cycle represented on Figure 4. Denoting the start of the interval as T, the best solutions obtained in the previous optimization interval serve as input to the *least cost insertion* module, which adds any new requests buffered during the previous cycle and outputs the best found solution. This is the solution to be deployed to the physical vehicles, $\dot{s}_{deployed}$, at time T', which would start to travel immediately accordingly to this route, ignoring any previous orders. Simultaneously to deployment, and while vehicles service their stipulated pickups and deliveries, another module called *end-state simulator* predicts where the vehicles will be at the end of the current time interval, based on how long the optimization interval is and how long each path takes to travel. This predicted route, $\dot{s}^*$, is used as input to the *static solver*, which will try to find improvements to this solution for $t_{ss} - (T' - T)$ minutes.

When the end of the time interval T^+ is reached, the *static solver* module is stopped. When dealing with static instances, the best solution is outputted from the module and it is terminated. In dynamic instances however, the best solution from the *static solver*, now $\dot{s}_{best}$, is grouped with the previously deployed solution, $\dot{s}_{best}$, and together they serve as input to next cycle's *least cost insertion model*, at T^+. All requests made available between T and T^+ are now introduced into the routes. The cycle is repeated until no more requests need to be serviced.

When solving dynamic instances, some changes are needed in the *static solver* module to account for requests that have already been fully serviced or whose pickup has already been serviced in the real world and thus is irreversibly tied to a vehicle. This means that the *new ant* module must also account for vehicle history when constructing the feasible node list. When generating new solutions with the local search strategy, any time a feasibility check is done it needs to also take into account nodes outside the current planned route but which are on the vehicle history.

Fig. 4. Schematics for Dynamic Solver cycle



The module *system end state* predicts where vehicles will be and what nodes have serviced during t_{ss}, time according to current distance matrix costs. To account for changes between the predicted end-state environment and the real environment at T', another module is needed to check if at the interval's end the predicted state matches the real state and fix anything needed accordingly. For the benchmark problems, the distance between nodes is computed using the Euclidean distance but for the case study it is calculated based on the Haversine formula.

G. Validation

To validate obtained results, the benchmarks for a pickup delivery problem with time windows instances generated by Li & Lim [11] will be used. They can be found on [1], under the PDPTW section, as well as the best results for each file found, which shall be referred to as the optimal solutions for each respective instance. The datasets for the dynamic problem will be generated from these files.

Since the presented results do not allow lateness and have as objectives: 1) minimization of vehicle number; 2) minimization of travelled distance and hard time windows, weights must be adjusted accordingly. This will be implemented by having the scaling factors such that $\mathcal{M}_4 \gg \mathcal{M}_2 > \mathcal{M}_1 \gg \mathcal{M}_3$.

Using a much greater factor for M_4 proved to be enough to steer away from any lateness in the solutions. The scaling factor for the waiting times, M_3, is given a very low influence so that our solutions incur in extra vehicles or travelled distance to decrease waiting times.

Benchmark file names contain encoded information on the type of problem. Using as example the 100 customer category (file name of type "lc1xx"), each file is labelled in accordance to the following logic:

topsep=0pt

- **lc**: generated requests are clustered geographically
- **lr**: generated requests are randomly distributed geographically
- **lcr**: generated requests are partially clustered and partially randomly distributed
- **1**: time windows have short and conflicting scheduling horizon (many vehicles)
- **2**: time windows have long scheduling horizon (few vehicles)
- **xx**: file identifier

IV. Results

A. Initial solution construction

To compare implementing the construction method with and without pre-clustering the customer nodes, Table I was generated. For each file and for each strategy it is present the average between all runs of: initial fitness value, the direct output of the constructor; run time at which the first solution without lateness appears; final fitness after 20 minutes. The best values in each case are highlighted in bold.

TABLE I
COMPARING INITIAL SOLUTION CONSTRUCTION USING NEAREST NEIGHBOURHOOD SEARCH WITH AND WITHOUT PRE-CLUSTERING WITH THE BEST VALUES FOR EACH COMPARISON IN BOLD

File name	Strategy	Initial Fitness	Transition Time (min)	Final Fitness
lc101	NNS	**188.58**	**0**	**182.89**
	Clustering	190.49	**0**	**182.89**
lc105	NNS	3.31E+11	0.44	89.16
	Clustering	**8.04E+10**	**0.07**	**88.89**
lc201	NNS	9.97E+09	8.04	437.11
	Clustering	**5.53E+09**	**2.41**	**310.02**
lc205	NNS	**347.24**	**0**	212.63
	Clustering	3.18E+10	1.73	**178.654**
lr101	NNS	**7.08E+09**	**2.54**	**348.73**
	Clustering	9.67E+09	2.824	355.91
lr105	NNS	**1.13E+11**	1.99	264.47
	Clustering	1.46E+11	**1.58**	**252.04**
lr202	NNS	**182.89**	**0**	**182.90**
	Clustering	190.49	**0**	**182.90**
lr205	NNS	3.18E+11	0.59	89.16
	Clustering	**1.89E+11**	**0.091**	**88.89**
lcr101	NNS	8.66E+09	6.42	421.25
	Clustering	**3.98E+09**	**2.19**	**309.89**
lcr102	NNS	2.44E+11	**1.74**	225.76
	Clustering	**6.02E+10**	2.02	**173.84**
lcr201	NNS	**7.33E+09**	3.57	367.03
	Clustering	1.06E+10	**2.60**	**359.92**
lc205	NNS	**9.6E+10**	2.06	275.44
	Clustering	1.48E+11	**1.42**	**247.83**

For both cases, the initial fitness show no special trend. The first time a solution with no lateness is generated, the *transition time* is either almost identical for both strategies or the pre-clustering ones are at least three times lower than only using NNS, trend which happens for half of the cases. So far, only a slight advantage is show for pre-clustering instead of only doing the NNS. However, in the majority of cases using NNS on pre-clustered points provides a better final fitness. Pre-clustering the requests and then applying a NNS to each cluster will be the strategy used in all other sections.

B. Static instances

Table II presents the average values for the error tables per type of file. Clustered data behaves differently from the others files as it manages to always reach the optimal number of vehicles for type 2 files. While for type 1 it does not reach the optimal value for all of them, it reaches a lower distance than the given by the optimal. For the other two the conclusions are similar, with average vehicle number error of 2 or less. Distance does fall below the optimal value as with the clustered files, but instead has an average error around 20%, which is alarmingly high.

Fig. 5. Optimal route plotting for the *lc101* file

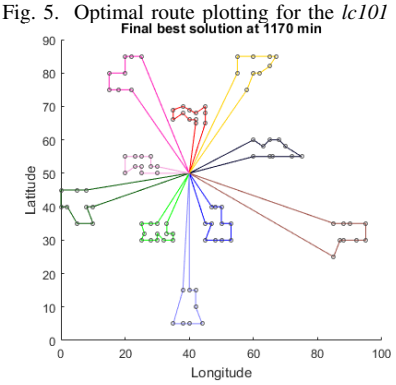


TABLE II
AVERAGE RESULTS OF MODEL VERSUS OPTIMAL VALUES FOR EACH FILE TYPE.

File	Vehicle				Distance			
	Mean	%	Best	%	Mean	%	Best	%
lc1	0.31	3.46	0.22	1.39	-41.49	-4.06	-23.79	-2.00
lc2	**0**	**0**	**0**	**0**	5.38	0.91	0.96	0.16
lr	0.18	1.83	0.12	1.31	-19.44	-1.72	-12.14	-0.98
lr1	1.39	12.99	0.67	6.57	131.87	11.33	58.50	4.99
lr2	1.05	40.54	0.55	21.96	329.41	35.31	168.63	18.15
lr	1.23	26.17	0.61	13.93	226.34	22.80	111.17	11.29
lcr1	2	16.62	1.5	12.42	161.70	7.34	98.89	2.84
lcr2	1.35	38.51	1	29.16	371.76	28.86	226.77	15.66
lc	1.68	28.93	1.25	21.83	266.73	17.86	162.83	9.07

C. Dynamic Instance

For dynamic testing, a problem instance will be generated based on *lc101*, chosen since the *static solver* is able to reach its optimal solution under 5 seconds. This way we are focusing only on testing the dynamic changes and the least insertion procedure by removing all difficulty for the *static solver* to find a solution. By looking at Figure 5 it is clear that using a clustered set of nodes makes for the demonstration makes it more intuitive to visualize the routes and the current solution quality.

Since no quantitative comparison can be made between dynamic and static solution we use this next section as a demonstration exercise, showing that the model works for the purpose it was created.

For simulation purposes it is necessary to distinguish between the actual run time of the static solver, t_{ss}, and the perceived time advanced by the algorithm, t_{sim}.

For each time the *static solver* runs for $t_s s$ time, the algorithm will see as if the time passed was actually t_{sim}.

For the dynamic test an important variable needs to be defined, the *lookahead* parameter, t_{look}. The generation of dynamic files consists in making nodes available to be serviced only after a specified time. Up to that point they are not accounted for in the route planning. This particular time is defined for each request as the earliest service time (so the a_i of the respective time window limit) of both the pickup and the delivery. In other words, a pickup-delivery pair is inserted into the set of customers to serve as soon as any of the two have their time window starting before the current simulation time, $t_{current}$, plus the lookahead parameter, which can be represented as when the condition $a_i < t_{current} + t_{look}$ is true.

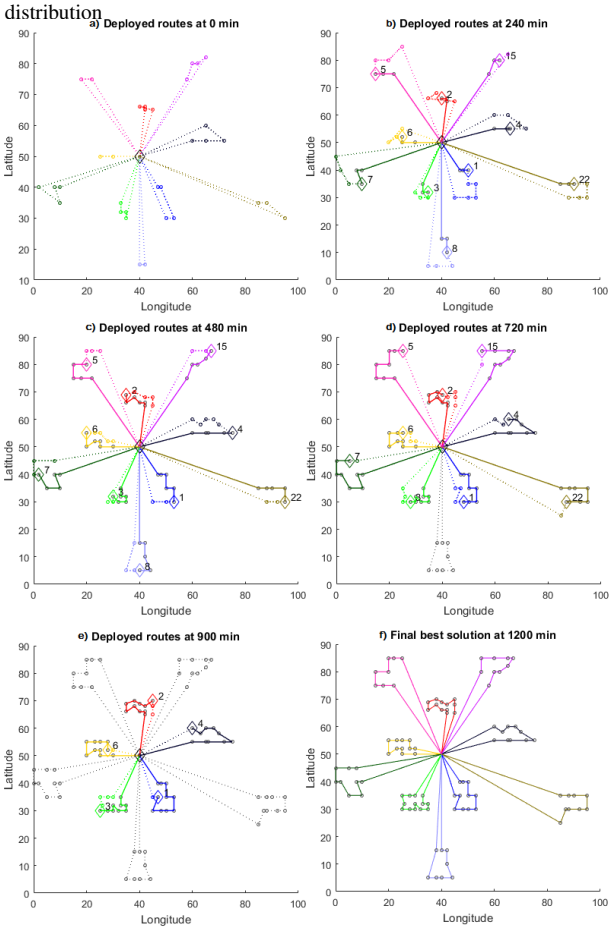
For dynamic testing, a problem instance will be generated based on *lc101*, chosen since the *static solver* is able to reach its optimal solution under 5 seconds. This way we are focusing only on testing the dynamic changes and the least insertion procedure by removing all difficulty for the *static solver* to find a solution. By looking at Figure 5 it is clear that using a clustered set of nodes makes for the demonstration makes it more intuitive to visualize the routes and the current solution quality.

On the visualizations of the simulation, as seen on Figure 6, for each vehicle their current location is represented by a diamond shape, each have a unique color and when outside of the depot are associated to their unique ID number. A dashed line represents the planned route for a given vehicle while a solid line represents the path previously travelled. Grey lines represent already serviced routes whose respective vehicles are back at the depot and circles represent customer nodes.

Here is presented a solution for the *lc101* file with $t_{look} = 45$ min, a $t_{static} = 1$ min and $t_{sim} = 15$ min. For this specific file, service time at the nodes is 90 minutes for almost all cases, being 0 on the other few. On Figure 6 we have plotted successive time instances, representative of the simulation's temporal evolution.

Starting at zero time, Figure 6 a) shows the planned route for the few initial nodes to service from the beginning. Next on Figure 6 b) is depicted the system state after 240 minutes, where vehicles have already moved from the depot and some customer nodes have been visited. At Figure 6 c), in the 480 minute mark, almost all requests have been inserted and vehicles have travelled through some more customers. On Figure 6 d) the first vehicle completes its route, and his previously travelled path is displayed in grey. At Figure 6 e), minute 900, almost all nodes have been serviced and only 5 vehicles remain active. Finally at Figure 6 f) all routes are completed, with all nodes serviced and the vehicles

Fig. 6. Visualization of dynamic routing for requests, geographical distribution



back at the depot. For this case, the achieved routes match the corresponding static solution.

The LCI module is responsible for adding new requests to the current routes at the end of each t_{ss} interval. Since insertion order is defined by the earliest time windows of a request, the LCI might lead to sub-optimal insertions, . If t_{sim} and t_{look} parameters are adequate to the current problem instance, the recently added nodes will only be visited after t_{ss} time, meaning they will pass at least once through *static solver* method. LCI is not designed to find the optimal insertion, but instead to provide a good and fast response, giving higher priority to the most urgent nodes first. This means a selected insertion move might be sub-optimal, such as depicted on Figure 7. Even with well adjusted t_{ss} and t_{look}, a recently added node at a sub-optimal position can be serviced right away, and thus become immutable. When such move is implemented, it is the *static solver*'s job to find the best possible solution given the visited node history. If t_{sim} and t_{look} parameters are adequate to the current problem instance, the chance of adding new

Fig. 7. Example of a sub-optimal LCI insertion

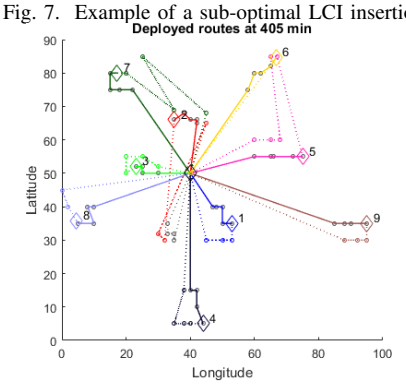


Fig. 8. Visualization of dynamic routing for requests, distribution of time in minutes per vehicle

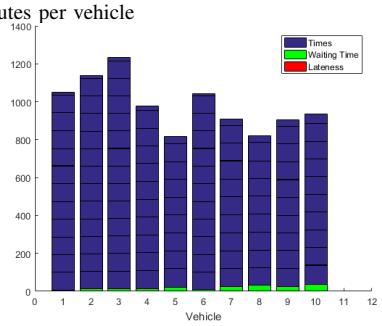


Fig. 9. Static solution for the case study, geographical distribution and service times plot

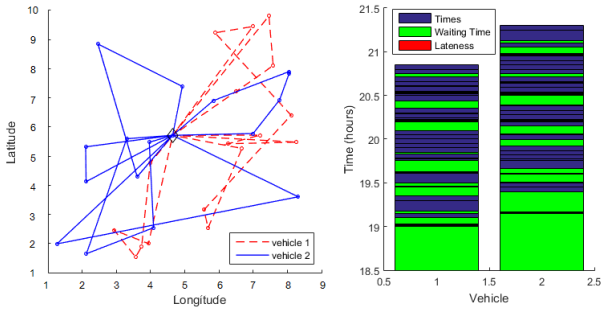
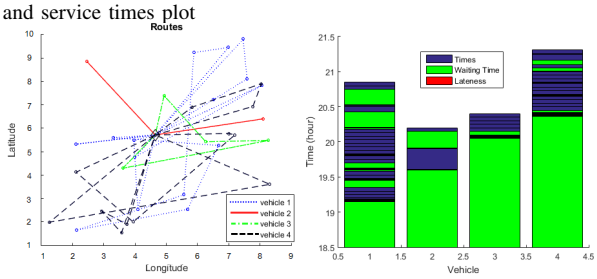


Fig. 10. Dynamic solution for the case study, geographical distribution and service times plot



nodes to route and service them without passing the *static solver* first decreases.

Lastly, it is presented the final time distributions per vehicles at Figure 8. Each bar is a unique vehicle and each segment a travel between two locations. As the label says, blue bars represent the vehicle travelling, green bars identify the vehicle waiting at location and red bars, if any, represent lateness of a delivery.

D. Case Study

The presented case study was proposed by the company UDD, together with most design constraints detailed next. The data is from a restaurant distribution service, meaning all pickups happen at the depot location. Due to the restaurant's nature, their current assumption for this data is that after a customer makes an order, the delivery time window starts in 45 minutes, which lasts for 15 minutes.

Currently, the distances between customer coordinates are calculated using the Haversine formula as a poor approximation to the actual road distances a vehicle has to travel using the traffic network. However, the created model can work with any matrix giving the distances and travel times between nodes. This means that instead of the Haversine formula one could easily switch to a better alternative, for example the *Google Maps Distance Matrix API*, and apply this model directly to a live problem. The provided data contains 47 requests to serve from 18:30h to 23h. Since an optimal solution for the tested data is not known, demonstration will be done comparing the dynamic run with the best achieved static solution.

First, the data is processed by the *static solver* module for 30 minutes, similarly to the approach on the static benchmarks. This will give a solution to be considered as the optimal when performing any dynamic tests. The case study data is similar to the benchmarks in size, with 47 requests to service. In terms of scheduling horizon, the case study matches the type 2 files. Visualization of a solution found can be seen on Figure 9.

For the static run, 2 vehicles were able to service all requests without lateness for a total of 51.4 km travelled. For the dynamic solution, with a $t_{sim} = 15$ minutes and a $t_{look} = 45$ minutes the solution obtained is represented in Figure 10. It uses 4 vehicles instead of 2 and has a total travelled distance of 68.6 km. It also arrives late at one location, but only less than a second after time window end.

V. Conclusions

A. Conclusions

The main objective was accomplished and a functional model was created to solve Capacitated Pickup Delivery Vehicle Routing Problems with Time Windows on a Dynamic Environment. Further, the proposed approach is

suitable for implementation in a real world environment, being able to deal with the tight time windows available to solve such heavily constrained problems. In general, the proposed approach shows a good performance in the validation benchmarks. The introduction of the initial clustering step improved the overall results of the proposed approach, only little improvements are needed to consider them competitive with other approaches from the state of the art.

Considering the benchmark problems where the data is not clustered, the proposed approach does not match the competition when comparing with other multi-vehicle pickup delivery problems, such as the one presented in [27], a commercially available software developed over 20 years by researchers or [32], where the benchmarks are introduced for validation of Pickup Delivery Problems with Time Windows. Even so, for at least two of these benchmarks is still able to find the optimal solution, which means that it is a viable approach. Under the case study presented, the proposed approach is able to solve the problem, presenting a solution without any delays in the delivery time windows.

The developed strategy for initial solution construction seems very promising and worth exploring further outside of this thesis. Pairing the ACO with Local Search proved to be a very good strategy since each method compensates for what the other is lacking. Nevertheless, the overall implemented model is slower than the initial model, due to the local search procedure, but the overall results are better. The used Local Search method really slows down the problem due to a feasibility check that runs for every generated route to the point where more than two thirds of the time is spent doing feasibility checks.

B. Future Work

The proposed model can be improved by adding the ability to handle different types of vehicles and to account for the actual distances and times needed to travel through the roads of a city in between the nodes location. Further, solutions for the case study should be provided to correctly assess what has already been developed and to see if it behaves better or worse than industry solutions.

One of the difficulties of the proposed approach is when the data is unordered, which means that the model of the static solver should be improved to provide better solutions. The first attempt to improve the algorithm would be to further elaborate the Local Search method, namely adding more diversity to different types of solution disturbance. As for the dynamic approach, besides the already mentioned usage of a more complete distance and time matrices and handling a heterogeneous fleet correctly, it is important for a model applied to a real case to refuse new requests if they will badly influence the already accepted routes. Using a more advanced waiting strategy could also lead to better solutions.

REFERENCES

[1] SINTEF Applied Mathematics. Transportation Optimization Portal - TOP, 2008. URL *https://www. sintef.no/projectweb/top/pdptw/li-lim-benchmark/*

[2] K. Steiglitz and C. H. Papadimitrou. Combinatorial Optimization: Algorithms and Complexity. Dover Publications, Mineola, New York, 1982..

[3] M. M. Flood. The Traveling-Salesman Problem. Operations Research, 4:6175, 1956.

[4] G. B. Dantzig and J. H. Ramser. The Truck Dispatching Problem Stable. 6(1):8091, 1959.

[5] K. Braekers, K. Ramaekers, and I. V. Nieuwenhuyse. The vehicle routing problem: State of the art classification and review. Computers & Industrial Engineering, 99:300313, 2016

[6] J. R. Montoya-torres, J. Lpez, S. Nieto, H. Felizzola, and N. Herazo-padilla. A literature review on the vehicle routing problem with multiple depots. Computers & Industrial Engineering, 79:115129, 2015. ISSN 0360-8352

[7] S. Mitrovi c-Mini c, R. Krishnamurti, and G. Laporte. Double-horizon based heuristics for the dynamic pickup and delivery problem with time windows. Transportation Research Part B: Methodological, 38(8):669685, 2004.

[8] L. M. Gambardella. Coupling Ant Colony System with Local Search. PhD thesis, 2015.

[9] Scholarpedia. Ant Colony Optimization, 2007. URL http://www.scholarpedia.org/article/ Ant_colony_optimization.

[10] G. . Hasle, K.-A. . Lie, and E. Quak. Geometric Modelling, Numerical Simulation, and Optimization: Applied Mathematics at SINTEF. 2007. ISBN 9783540687825.

[11] H. Li and A. Lim. A Metaheuristic for the Pickup and Delivery Problem with Time Windows.

[12] F. Ferrucci and S. Bock. Real-time control of express pickup and delivery processes in a dynamic environment. Transportation Research Part B: Methodological, 63:114, 2014.

[13] D. S. Johnson and L. A. Mcgeoch. The Traveling Salesman Problem : A Case Study in Local Optimization. In Local Search in Combinatorial Optimization, pages 215310. 1997.

[14] L. Bianchi. Ant Colony Optimization And Local Search For The Probabilistic Traveling Salesman Problem: A Case Study in Stochastic Combinatorial Optimization. PhD thesis, 2006.