Dynamic Multiple Vehicle Routing under Energy Capacity Constraints*

Giorgos Polychronis[1] and Spyros Lalis[2]
University of Thessaly

***Abstract*— The multiple vehicle routing problem (mVRP) concerns the scheduling of multiple vehicles so as to visit some locations of interest. We study a dynamic version of mVRP where the travel costs are not a priori known and may vary at runtime. Moreover, we introduce energy-related constraints which make the problem more complex. Vehicles have only finite energy reserves, which gradually diminish as they move between different locations, but can also gain some energy at specific depot locations. The objective is to visit all locations of interest as fast as possible without any vehicle exhausting its energy. We propose an online algorithm based on the Large Neighbourhood Search (LNS) heuristic. We evaluate the algorithm for different topologies and degrees of vehicle autonomy. Our results show that it achieves significantly better results than an offline algorithm that produces a safe schedule based on worst-case cost estimates.**

I. INTRODUCTION

Unmanned vehicles (UVs) are becoming more popular and have the potential to revolutionize several civilian application domains, such as agriculture, transport, surveillance. A common mission pattern is for UVs to visit locations of interest in order to perform various sensing/actuation tasks. When planning such missions, one must take into account the distances that have to be covered by the UVs in order to reach the different locations of interest as well as the time and energy that is needed for this.

This problem corresponds to the multiple vehicle routing problem (mVRP), which has been studied in many forms and for different optimization objectives. Particularly challenging are the dynamic variants of mVRP, where new locations of interest may appear while the planned trip is still in progress, or the cost of travel between the locations to be visited may differ from what was initially assumed during planning.

In this paper, we also tackle a variant of the dynamic mVRP where the travel costs can vary at runtime. We let the travel cost correspond to the energy spent by a vehicle in order to move between two locations, and introduce special depot nodes that can be used to restore the energy reserves of the vehicles. However, we introduce a hard limit for the maximum energy reserves and do not allow vehicles to deplete their energy at any point during the mission.

*This research has been co-financed by the European Union and Greek national funds through the Operational Program Competitiveness, Entrepreneurship and Innovation, under the RESEARCH-CREATE-INNOVATE call, project PV-Auto-Scout, code T1EDK-02435.

[1]Giorgos Polychronis is a PhD student at the ECE Department, University of Thessaly, Volos, Greece gpolychronis@uth.gr

[2]Spyros Lalis is with the Faculty of the ECE Department, University of Thessaly, Volos, Greece lalis@uth.gr

This formulation closely models the problem of managing a fleet of UVs that may need to refuel or change batteries in order to continue their mission. When a UV runs our of energy, it enters a safe mode, which may involve performing an emergency landing/parking action, and no longer participates in the mission.

The main contributions of this paper are as follows. (i) We describe the above problem in a formal way. (ii) We propose an online heuristic that adjusts a conservative initial schedule that is constructed based on worst-case cost estimates. (iii) We evaluate the algorithm for different topologies, degrees of uncertainty and vehicle autonomy.

The rest of the paper is structured as follows. Section II discusses related work. Section III gives the problem formulation. Section IV describes the proposed heuristic. Section V presents our evaluation. Finally, Section VI concludes the paper and points to some directions for further research.

II. RELATED WORK

The vehicle routing problem (VRP) has been studied for different dynamic aspects. Indicative surveys can be found in [1], [2], [3]. Most studies focus on the dynamic arrival of new targets, also commonly referred to as customers or customer requests. Some works consider dynamic travel times while others address the problem of known customers with uncertain service needs, which are revealed to the vehicles and the planning system when a customer is actually visited. Below, we provide an overview of indicative approaches that include the aspect of dynamic travel costs/times and are thus closer to the problem we tackle in this paper.

In [4], the authors address the problem where new customer requests arrive online and there is uncertainty regarding the travel times. Their online approach re-constructs the routes of the vehicles by using the insertion heuristic, every time a new demand arrives or when the travel times change. They further improve the solution with the Or-opt algorithm [5], where a segment of a route (a number of consecutive customers) is moved to a different position.

A genetic algorithm is proposed for the same problem in [6], where the routes of the vehicles are periodically adjusted taking into account the new information that becomes available regarding changes in the travel times and new customer requests. Another genetic algorithm is presented in [7], which is used to produce the initial schedule as well as to perform any adaptations at a later point in time. In this case, rescheduling occurs when the estimated travel times for an edge is updated (once a vehicle reaches a customer) with the latest information from a dynamic traffic simulator.

 DOI: 10.1109/ITSC45102.2020.9294492

In the approach described in [8], every vehicle is allowed to visit the next customer with some tolerance for delays beyond the expected arrival times. If this tolerance is exceeded, the customer is removed from the current route and is reinserted in the best position in the route of another vehicle. In a continuation of this work, the customer can also be added back to the route from where it was removed [9]. In both cases, the paths after the insertions are further improved with the CROSS exchange algorithm, which is proposed in [10] for exchanging entire segments (consecutive customers) between routes. As a further extension, [11] considers a continuous tracking of the vehicle, where unexpected delays are detected earlier, before the tolerance is actually exhausted.

The authors of [12] also deal with the problem of newly arriving requests, in combination with varying travel speeds. In this case, the speed of the vehicle is known as soon as it starts its trip towards the next customer. The routes are adapted using four metaheuristics. The first two approaches are based on a dynamic Variable Neighbourhood Search (VNS) algorithm, based on static and stochastic information, respectively. Two additional metaheuristics are presented, using a Multiple Scenario and a Multiple Plan approach, where multiple solutions are pursued/maintained in parallel, based on static and stochastic costs, respectively. Both approaches use as a search procedure the VNS algorithm.

A wider range of dynamic events is considered in [13], each one possibly necessitating an adaptation of the scheduled routes. More specifically, these events are: late arrival of a vehicle at a customer; timely arrival of the vehicle at a customer that is no longer valid; cancellation of a customer request; generation of a new customer request; break-down of a vehicle; vehicles being stuck in a traffic jam. Two operators are used to optimize the schedule. In the first case, individual customers are removed from a route and it is attempted to insert them in another route. In the second case, customers are exchanged in a pairwise fashion between routes. The customer insertions are done greedily. The authors also use a secondary objective function to drive the local search so that it focuses on more promising neighbourhoods.

The main difference of our work is that it places a hard constraint on the energy capacity of the vehicles. As a consequence, dynamic changes in travel costs have an impact not only on the optimality but also on the feasibility of the schedule. Also, any adaptations that are made to optimize the routes of the vehicles, must take this constraint into account in order to produce schedules that are actually feasible.

Dynamic routing has also been studied specifically for drone-based applications. For instance, [14] presents a delivery scenario where a fleet of drones is used to ship meals to locations that become known at runtime. The drones have payload carrying capacity and energy constraints, and can swap batteries at depot stations. The objective is to avoid having drones that run out of energy and to minimize delivery times and unnecessary flights. A similar problem is studied in [15] for a heterogeneous fleet of drones, taking into account the payload capacity, the energy capacity and the maximum speed of each drone. Although these problems also introduce hard energy constraints, they do not deal with uncertainty regarding the energy consumption of the drones.

III. PROBLEM FORMULATION

A. Terrain

We model the terrain where the mission takes place as a directed graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$, where $\mathcal{N}$ is a set of nodes and $\mathcal{E}$ is a set of edges. Each node $n_i \in \mathcal{N}$ represents a target location that has to be visited by a vehicle. Each edge $e_{i,j} \in \mathcal{E}$ represents a possible movement from node n_i directly to node n_j, without going via any intermediate nodes.

B. Energy capacity, costs and gains

The vehicles used to visit the nodes have a finite energy storage capacity B. This can be thought of as the capacity of a fuel tank or the capacity of a battery, depending on whether the vehicles are equipped with internal combustion engines or electrical motors. Let $b \leq B$ denote the current energy budget (reserves) of the vehicle during travel.

When the vehicle moves between two nodes, its motors consume some of the available energy budget. Let $c_{i,j}$ denote the cost for moving from n_i to n_j over $e_{i,j}$, also referred to as edge cost. If the vehicle has an energy budget b and moves from n_i to n_j over $e_{i,j}$, the remaining energy budget will be $brem = b - c_{i,j}$. If $brem \leq 0$, the vehicle exhausts its energy and becomes non-operational before reaching n_j.

The vehicle may increase its energy budget by gaining energy at depot nodes, which can be refuelling or battery switching stations. Let g_i be the energy that can be gained at n_i. If n_i is not a depot, $g_i = 0$.

C. Path feasibility

A path p is encoded as a sequence of nodes, where $p[k], 1 \leq k \leq |p|$ is the k^{th} node in p and $|p|$ is the number of nodes in the path. Equivalently, let $e_{i,j} \in p$ if $p[k] = n_i$ and $p[k+1] = n_j$ for $1 \leq k \leq |p| - 1$. Note that if p is non-empty then $|p| \geq 2$. Also, let $p[k1 : k2]$ denote the part of the p that starts from node $p[k1]$ and ends at node $p[k2]$.

Let $remb(b, p)$ denote the remaining budget of the vehicle if it starts with an initial budget b and travels along path p. This can be expressed as follows:

$$remb(b,p) = \begin{cases} min(B, b + g_{p[1]}) - c_{p[1],p[2]}, & |p| = 2 \\ remb(remb(b, p[1:2]), p[2:|p|]), & |p| > 2 \end{cases} \quad (1)$$

Namely, if p consists of a single hop, the remaining budget is equal to the initial budget plus the energy gain (if any) at the start node $p[1]$ less the edge cost for moving from $p[1]$ to $p[2]$. Note that the gain at the destination node $p[2]$ (if any) is not taken into account as this cannot be used to perform the hop in question. If p includes more than one hops, the remaining budget at the end of p is equal to the remaining budget for the path without the first hop, starting with a budget that is equal to the remaining budget after taking the first hop. In this case too, the remaining budget at the end of the path does not include the gain at the destination node.

Based on the above, we say that a path p is feasible for an initial budget b, if $remb(b, p[1:k]) > 0, 1 < k \leq |p|$. In other words, p is feasible if the vehicle will not exhaust its energy budget at any point along p. Also, let $nodes(p)$ denote the set of nodes that are part of p.

D. Path completion time and schedule makespan

Let $travel()$ be a function that transforms edge energy costs to the corresponding travel times. Then, the time needed to complete path p is $d(p) = \sum_{e_{i,j} \in p} travel(c_{i,j})$. Notably, we assume that depot service times are negligible compared to the travel times, which is realistic for conventional refuelling as well as for battery switching stations.

Let a schedule $s[]$ consist of M paths $s[m], 1 \leq m \leq M$. Assuming that each of these paths can be pursued in parallel, the makespan of s is $max_{m=1}^{M} d(s[m])$. In other words, it is equal to the completion time of the most costly path.

E. Problem statement

Let there be M vehicles that can travel independently. The objective is to find a feasible schedule (a schedule that consists of feasible paths), such that the vehicles will visit all the nodes of interest with the smallest possible delay. More formally, $s[]$ should be chosen so that $\cup_{m=1}^{M} nodes(s[m]) = \mathcal{N}$ and $max_{m=1}^{M} d(s[m])$ is minimized.

In this work, we focus on a dynamic version of the problem. Namely, the edge/travel costs are unknown to the fleet scheduler and may vary in time. More specifically, we model the edge cost $c_{i,j}$ as a random variable over the range $[c_{i,j}^{min} .. c_{i,j}^{max}]$ with an expected/mean value of $c_{i,j}^{mean}$. This problem cannot be addressed in a satisfactory way using an offline algorithm. Instead, an online algorithm is needed in order to cope with the system dynamics.

We assume that depot nodes are known in advance and can always restore the vehicles' energy reserves to the maximum capacity B. We let all vehicles start from a depot node, and require that they also return back to a depot node. The makespan applies to the successful completion of the mission: all nodes have been visited and all vehicles have safely returned to a depot. Last but not least, we assume no dead-ends, that is, the maximum edge costs $c_{i,j}^{max}$ and maximum capacity B are so that each node can be visited by a vehicle that starts from a depot and then has enough energy to return back to a depot.

IV. ALGORITHM

A. Representation of schedules and state information

We represent the schedule and state of a given vehicle as a tuple $(paths[], remb_{est}[], remb_{upd})$, where $paths[]$ is the list of paths that have been assigned to it. Each path starts from a depot node and end at a depot node, thus the vehicles always start a new path with the maximum energy budget B. Note that the end depot node of $paths[i]$ is the start depot node of $paths[i+1]$. The estimated remaining budget after having completed $paths[i]$ is stored in $remb_{est}[i] = remb(B, paths[i])$.

The path that is currently followed by a vehicle is always the first path in the list $paths[1]$. When this path is completed, it is removed from the list, and the vehicle begins to follow the next (first) path in the list. Each time the vehicle performs the next hop along the current path, it experiences the actual cost for this movement. This can be different than what was estimated when that path was planned. To keep track of this deviation, the updated estimate for the remaining budget at the end of the current path is stored in $remb_{upd}$.

The complete schedule and state information $s[]$ is a list of M tuples, where $s[m], 1 \leq m \leq M$ corresponds to the m^{th} vehicle that is used to visit the nodes of interest.

B. Cost estimation

There are different ways to estimate the remaining budget when a path is planned or it is updated. One policy could be to use the average / expected edge costs. Another policy would be to use the maximum possible (worst-case) costs.

Optimistic approaches may lead to better schedules with a smaller makespan, but they also introduce the risk of vehicles exhausting their energy budget and becoming non-operational. This, in turn, can have a very negative impact to the mission, and lead to far worse results. Conservative approaches may generate less optimal schedules but reduce the probability of some vehicles becoming non-operational.

In this work, we explore the most conservative approach. Namely, we estimate the remaining budgets based on the maximum possible (worst-case) edge costs. This way it is guaranteed that the generated schedules are *always* feasible ($s[m].remb_{est}[i] > 0, \forall m, i$). In turn, this guarantees that all nodes will be visited and all vehicles will manage to return to a depot. Moreover, this means that any cost deviations concerning the current path of a vehicle will result in greater energy reserves than the ones estimated when that path was planned ($s[m].remb_{upd} > s[m].remb_{est}[1]$).

C. Main loop

The algorithm starts from an initial (feasible) schedule $s[]$. Then, the vehicles are instructed to commence the mission, and start following the paths that have been assigned to them in the order in which they appear in their path lists.

The high-level logic of the algorithm is shown in the form of pseudocode in Algorithm 1. Each iteration corresponds to the attempt of one or more vehicles to perform the next hop in their current path. If the actual cost for this movement is different than what was estimated when the path was planned, the state of the vehicle is updated, adjusting $s[m].remb_{upd}$ accordingly.

If this wasn't the last hop along the current path, it is checked whether there is a large relative deviation between the updated remaining budget for that path $s[m].remb_{upd}$ and the corresponding estimate $s[m].remb_{est}[1]$ that was made when the path was planned. If such a deviation exceeds a threshold, a flag is set. The threshold for this deviation is configurable.

The flag is checked after the state of all vehicles has been updated. If it is set, this triggers a rescheduling attempt. The

heuristic for this is abstracted via function LNS() and is discussed in more detail in the sequel. The rescheduling may change parts of the schedule $s[]$, adjusting the paths and state of the vehicles accordingly. Note that rescheduling does not change the current destinations of the vehicles, which always complete the current hop as planned.

The algorithm ends when all vehicles have completed the paths that were assigned to them. Recall that paths are planned in a conservative way, thus the schedules generated are always feasible and no vehicle will run out of energy during travel.

Algorithm 1 Online scheduling algorithm.

```
function SCHEDULER(s, threshold, nofmutations)
    while ∃m : s[m].paths[1] ≠ ∅ do
        reschedule ← false
        for each m performed k^th hop along its path do
            i, j ← s[m].paths[1][k], s[m].paths[1][k + 1]
            s[m].remb_upd ← s[m].remb_upd + (c^max_{i,j} − c_{i,j})
            if k ≠ |s[m].paths[1]| − 1 then
                dev ← s[m].remb_upd − s[m].remb_est[1]
                if dev/s[m].remb_est[1] > threshold then
                    reschedule ← true
                end if
            else
                s[m].paths[].pop()        ▷ remove first path
            end if
        end for
        if reschedule then
            s ← LNS(s, nofmutations)
        end if
    end while
end function
```

D. Reschedule heuristic

The reschedule optimization technique follows the principle of Large Neighbourhood Search (LNS), which was originally proposed in [16]. Its main advantage is that it can explore larger neighbourhoods than local search algorithms. Additionally, with the right removal/reinsertion functions, the search can check promising neighbourhoods in a fast way.

In our case, we use LNS to find a new schedule by performing a number of mutations to the current schedule. The high-level logic of the heuristic is given in Algorithm 2. Each mutation involves (a) the random removal of some nodes from the paths that are assigned to the vehicles, and (b) the reinsertion of the nodes that were removed into the schedule. In each iteration, the number of nodes to be removed / reinserted is picked randomly. The bounds for the respective interval are retrieved via function $getNodeBounds()$. We configure this function to return the interval $[\sqrt{N}..min(N, \sqrt{N} \times 4)]$, where N is the number of nodes that still need to be visited. If the mutation produces a better schedule (in terms of makespan), it is used as the basis for the next mutation. The number of mutations is configurable via the respective parameter.

Algorithm 2 LNS-based schedule optimization method.

```
function LNS(s, nofmutations)
    s^best ← s
    lower, upper ← getNodeBounds(s)
    repeat nofmutations times
        nofnodes ← random(lower, upper)
        s^tmp, free ← REMOVE(s^best, nofnodes)
        s^new ← INSERT(s^tmp, free)
        if makespan(s^new) < makespan(s^best) then
            s^best ← s^new
        end if
    end repeat
    return s^best
end function
```

We have designed the node removal function to work as shown in Algorithm 3. A number of so-called seed nodes is picked randomly from the interval retrieved via function $getSeedBounds()$. This is configured to return the interval $[1..max(1, \sqrt{N} \times 0.4)]$. As above, N is the number of nodes that still need to be visited. The specific seed nodes to be removed are chosen randomly out of all the nodes that still need to be visited. Then, each of the seed nodes along with some of its closest neighbours (nodes connected to it with the smallest edge cost) are removed. The number of seed neighbours to remove is chosen so as to have a balanced node removal around all seeds. In Algorithm 3, the node removal procedure is abstracted via the $removeNode()$ function, which also updates the internal schedule information accordingly. The removal method returns the modified schedule and the set of nodes that have been removed.

Algorithm 3 Node removal method.

```
function REMOVE(s, nofnodes)
    lower, upper ← getSeedBounds(s)
    nofseeds ← random(lower, upper)
    nofnodes' ← nofnodes/nofseeds
    seeds ← pickRandom(s, nofseeds)
    s^new ← s
    free ← ∅
    for each seed ∈ seeds do
        repeat nofnodes' times
            n ← fndNearestNode(s^new, seed) ▷ incl. seed
            s^new ← removeNode(s^new, n)
            free ← free + n
        end repeat
    end for
    return s^new, free
end function
```

The node reinsertion method, is described in Algorithm 4. A greedy heuristic is followed, whereby each node is inserted at the best (regarding the makespan) feasible position in the path of some vehicle. Function $findPos()$ abstracts

the procedure for finding the best possible position in the schedule of a given vehicle. It returns the suggested path with the node insertion, the path's position in the path list of the vehicle, the updated estimate for the makespan of the vehicle and the updated makespan for the entire schedule assuming the suggested insertion were adopted. Note that only feasible insertions are returned from this function.

Algorithm 4 Node insertion method.

```
function INSERT(s, nodes)
    s^new ← s
    while nodes ≠ ∅ do
        t^s_min ← ∞                          ▷ min schedule makespan
        t^v_min ← ∞                          ▷ min vehicle makespan
        n ← randomPick(nodes)
        for each m do
            p, k, t_v, t_s ← findPos(n, s^new[m].paths)
            if t^s < t^s_min then
                m^best, p^best, k^best ← m, p, k
                t^s_min, t^v_min ← t^s, t^v
            else if t^s = t^s_min ∧ t^v < t^v_min then
                m^best, p^best, k^best ← m, p, k
                t^v_min ← t^v
            end if
        end for
        s^new ← updatePath(s^new[m^best].paths[k^best], p^best)
    end while
    return s^new
end function
```

Each top-level iteration concerns the insertion of a single node. After checking all vehicles, the best insertion option is chosen for the given node: the one that leads to the smallest increase of the schedule makespan, or, in case of a tie, the one that leads to the smallest increase of the makespan for the vehicle that will visit the node. The path update process is abstracted via function $updatePath()$, which returns the new schedule with all state information accordingly updated.

If a node cannot be inserted in an existing path (due to the capacity constraint), a new path is created for visiting that node, and this is assigned to the vehicle with the smaller makespan. In this case, the new path is added at the end of the vehicle's path list. For the sake of brevity, this is not explicitly shown in the pseudocode; this special case is handled in a transparent way by the $findPos()$ and $updatePath()$ functions.

V. EVALUATION

A. Experimental setup

We evaluate the proposed algorithm using simulations. We conduct our experiments for a 11×11 grid of 121 nodes. The nodes represent the geographical locations of an area that has to be scanned by a team of vehicles exhaustively, by visiting all nodes. We assume that the vehicles can freely move between nodes in a straight line. This is typically the case for aerial unmanned vehicles (UAVs) that scan a large area from a relatively high altitude that keeps them safely above trees and power lines. Thus, there is an edge between every two nodes.

The cost of each edge $e_{i,j}$ represents the cost for moving from n_i to n_j. This cost is *not known* with certainty, but randomly varies following a uniform distribution $[c_{i,j}^{min}..c_{i,j}^{max}]$. The upper and lower bounds of the cost distribution are a function of the Euclidean distance between the nodes' positions. More specifically, $c_{i,j}^{min} = \alpha \times dist(n_i, n_j)$ and $c_{i,j}^{max} = \beta \times dist(n_i, n_j)$. We investigate scenarios for *small uncertainty* ($\alpha = 0.5$ and $\beta = 1$) and *large uncertainty* ($\alpha = 0.25$ and $\beta = 1$). Without loss of generality, we let the travel time be a linear function of the edge cost.

To capture the uncertainty of the travel cost, we create 50 different scenarios. In each scenario, the cost of every edge is randomly chosen based on the respective random distribution. The actual cost of an edge is discovered only at runtime, when a vehicle crosses that edge and reaches the destination node. Given that each node has to be visited once and that the graph is fully connected, every edge is traversed at most once, thus it suffices to have only one randomly chosen cost value for each edge.

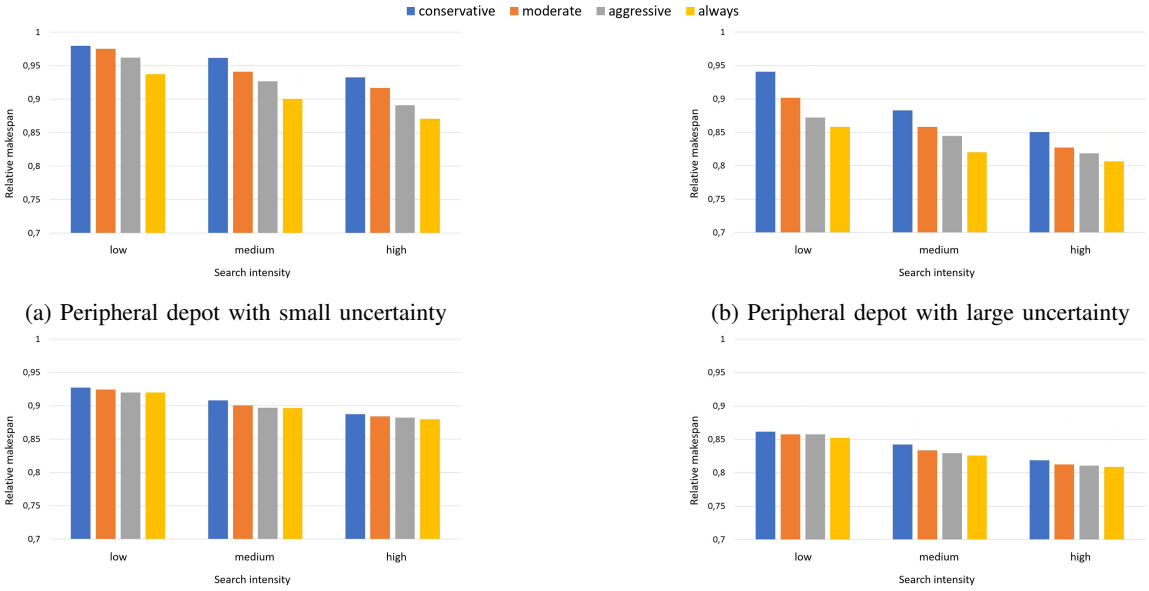
In our experiments we have a single depot node, and investigate two different scenarios regarding its location. In the *peripheral depot* scenario, the depot node is located at one of the corners of the grid. In the *central depot* scenario, the depot node is located at the center of the grid.

We use a fleet of $M = 3$ vehicles, which all have the same energy capacity. The capacity of the vehicles is set to the worst-case round-trip cost, between the depot node and the node that is farthest away from it. This ensures that it is indeed possible to visit all nodes, even if the edge costs turn out to be the maximum possible.

B. Configurations of the online algorithm and reference

We test the online algorithm for various configurations. On the one hand, we experiment with four rescheduling thresholds 0.2, 0.1, 0.05 and 0.0, referred to as *conservative*, *moderate*, *aggressive* and *always* configurations, respectively. Recall that lower thresholds lead to more frequent/earlier rescheduling. In the *always* configuration the algorithm reschedules whenever a deviation is experienced, irrespectively of how significant this is relative to the path cost. On the other hand, we vary the number of iterations in the LNS component, from $25, 50$ to 100 iterations. We refer to this as *low*, *medium* and *high* search intensity, respectively. Path changes do not change the current destination of a vehicle but may affect the remaining path(s). The algorithm executes quickly (under a second) thus it can run even after every hop without delaying the vehicles.

As a reference, we use a schedule that is produced offline. The algorithm we use for this is based on IWO [17], which provides good results for the min-max mTSP problem. We extended IWO to tackle the vehicle capacity and path feasibility constraints. The offline schedule is generated based on the worst-case cost of each edge, and thus is guaranteed to be feasible (no vehicle will ever run out of energy)



(a) Peripheral depot with small uncertainty

(b) Peripheral depot with large uncertainty

(c) Central depot with small uncertainty

(d) Central depot with large uncertainty

Fig. 1: Makespan of the online algorithm vs. the offline algorithm.

independently of the actual costs that will be experienced by the vehicles during the mission. The offline schedules are also used as the initial schedules for the online algorithm.

C. Results

We run each configuration of the online algorithm 5 times for each of the 50 edge cost scenarios (for each depot and uncertainty scenario). Figure 1 shows the average makespan of the different configurations as a percentage of the makespan of the offline algorithm. At the top, Figure 1a and Figure 1b show the results for the peripheral depot scenario, while Figure 1c and Figure 1d at the bottom show the central depot scenario. The figures on the left show the results for the small uncertainty scenario, and the ones on the right for the large uncertainty scenario. For the two "extremes" of our experimental study, the improvement varies from 2% for the most conservative rescheduling and low search intensity configuration in the peripheral depot node scenario with small uncertainty, up to nearly 20% for the most aggressive rescheduling and highest search intensity configuration in the central depot node scenario with large uncertainty.

Increasingly better results are achieved when the online algorithm reschedules more aggressively (more often). This is because the sooner one replans during the mission, the more likely it is to achieve a significant optimization in the routes of the vehicles as there are still many unvisited nodes. More conservative (less frequent) rescheduling misses such opportunities. Even though the accumulated reserves are larger in this case, rescheduling takes place after having visited several nodes, so there are fewer opportunities for major optimizations. Also, as expected, better results are achieved for the more intensive search configurations.

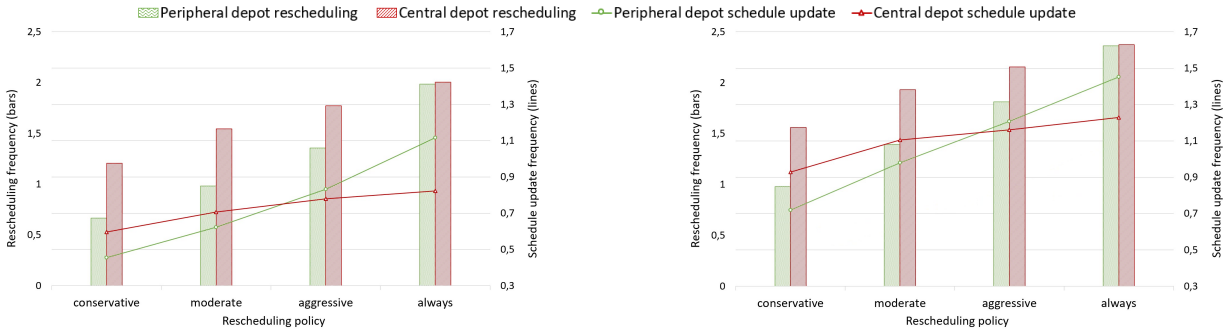
The schedule improvements are better under larger cost uncertainty. Given that path planning is based on worst-case estimates, any scheduling decisions become increasingly sub-optimal under larger uncertainty, and this can only be repaired by rescheduling more often.

Note that the improvements are more significant in the central depot scenario for the conservative, moderate and aggressive reschedule policies. The reason is that the paths are shorter so the respective worst-case estimates are smaller than in the peripheral scenario. As a consequence, the same absolute cost deviations meet the respective thresholds more often, leading to more frequent rescheduling attempts. The always reschedule policy achieves an equally good improvement in both depot scenarios. In this case, a rescheduling attempt is done at the slightest cost deviation (even if minor compared to the estimated path costs), thus rescheduling is performed equally frequently in both scenarios.

Figure 2 shows the rescheduling and schedule update frequencies, averaged over all search intensity configurations. These are calculated by dividing the total number of rescheduling attempts and respectively the total number of actual schedule updates that were performed during the mission, by the makespan. In practical terms, this corresponds to the average number of rescheduling attempts / schedule updates performed while a vehicle is travelling between two nodes; values can be larger than 1 as there are many vehicles that travel concurrently over different distances.

Naturally, the rescheduling frequency (bars) increases with more aggressive rescheduling and for larger uncertainty. As discussed above, the conservative, moderate and aggressive policies (non-zero thresholds) lead to more frequent rescheduling in the central than in the peripheral depot scenarios, while the always reschedule policy (zero threshold) leads to practically the same rescheduling frequency.

The more rescheduling attempts are made, the more likely it is that at least some of them will succeed, leading to a



(a) Small uncertainty

(b) Large uncertainty

Fig. 2: Rescheduling and schedule update frequency of the online algorithm (average over all search intensity configurations).

higher schedule update frequency (lines). Note that conservative and moderate rescheduling leads to a higher update frequency in the central vs. the peripheral depot scenarios, whereas the situation is reversed in the aggressive and always rescheduling policies. Rescheduling attempts are in general more likely to succeed in the peripheral depot scenarios because paths are longer hence the worst-case estimates are also more likely to be overly pessimistic. Therefore, as the gap of the rescheduling frequency between the two scenarios shrinks, the schedule update frequency becomes higher in the peripheral depot scenario. Still, the impact of each individual update is less significant in the peripheral depot scenario, which is the reason why practically the same improvement is achieved in both cases with the always reschedule policy.

While the proposed heuristic clearly outperforms the static scheduling, it is far from optimal. An oracle with a priori knowledge of the costs that will be experienced during the mission, can further reduce the makespan by 25% up to 45% depending on the scenario. So, there seems to be plenty of room for optimizations, provided one is willing to risk that some vehicles may not be able to safely return to a depot.

VI. CONCLUSIONS

We presented an online algorithm for tackling the mVRP for uncertain travel costs and vehicles with hard capacity constraints, based on an LNS component for schedule updates. Starting from a conservative offline schedule, cost deviations that occur at runtime are exploited to let vehicles visit more nodes before returning to a depot. Our experiments show that this can reduce the makespan significantly, especially when there is large uncertainty regarding the travel costs.

We plan to examine different topologies and node removal variants. We also wish to investigate algorithmic extensions to handle non-negligible depot service times and to explore more optimistic planning heuristics in conjunction with methods for learning from previous travel experience.

REFERENCES

[1] U. Ritzinger, J. Puchinger, and R. F. Hartl, "A survey on dynamic and stochastic vehicle routing problems," *International Journal of Production Research*, vol. 54, no. 1, pp. 215–231, 2016.

[2] H. N. Psaraftis, M. Wen, and C. A. Kontovas, "Dynamic vehicle routing problems: Three decades and counting," *Networks*, vol. 67, no. 1, pp. 3–31, 2016.

[3] V. Pillac, M. Gendreau, C. Guéret, and A. L. Medaglia, "A review of dynamic vehicle routing problems," *European Journal of Operational Research*, vol. 225, no. 1, pp. 1–11, 2013.

[4] H.-K. Chen, C.-F. Hsueh, and M.-S. Chang, "The real-time time-dependent vehicle routing problem," *Transportation Research Part E: Logistics and Transportation Review*, vol. 42, no. 5, pp. 383–408, 2006.

[5] I. Or, "Traveling salesman-type combinatorial problems and their relation to the logistics of blood banking," *PhD thesis (Department of Industrial Engineering and Management Science, Northwestern University)*, 1976.

[6] A. Haghani and S. Jung, "A dynamic vehicle routing problem with time-dependent travel times," *Computers & Operations Research*, vol. 32, no. 11, pp. 2959–2986, 2005.

[7] E. Taniguchi and H. Shimamoto, "Intelligent transportation system based dynamic vehicle routing and scheduling with variable travel times," *Transportation Research Part C: Emerging Technologies*, vol. 12, no. 3-4, pp. 235–250, 2004.

[8] J.-Y. Potvin, Y. Xu, and I. Benyahia, "Vehicle routing and scheduling with dynamic travel times," *Computers & Operations Research*, vol. 33, no. 4, pp. 1129–1137, 2006.

[9] S. Lorini, J.-Y. Potvin, and N. Zufferey, "Online vehicle routing and scheduling with dynamic travel times," *Computers & Operations Research*, vol. 38, no. 7, pp. 1086–1090, 2011.

[10] É. Taillard, P. Badeau, M. Gendreau, F. Guertin, and J.-Y. Potvin, "A tabu search heuristic for the vehicle routing problem with soft time windows," *Transportation Science*, vol. 31, no. 2, pp. 170–186, 1997.

[11] J. Respen, N. Zufferey, and J.-Y. Potvin, "Online vehicle routing and scheduling with continuous vehicle tracking," 2014.

[12] M. Schilde, K. F. Doerner, and R. F. Hartl, "Integrating stochastic time-dependent travel speed in solution methods for the dynamic dial-a-ride problem," *European Journal of Operational Research*, vol. 238, no. 1, pp. 18–30, 2014.

[13] Z. Xiang, C. Chu, and H. Chen, "The study of a dynamic dial-a-ride problem under time-dependent and stochastic environments," *European Journal of Operational Research*, vol. 185, no. 2, pp. 534–551, 2008.

[14] Y. Liu, "An optimization-driven dynamic vehicle routing algorithm for on-demand meal delivery using drones," *Computers & Operations Research*, vol. 111, pp. 1–20, 2019.

[15] B. N. Coelho, V. N. Coelho, I. M. Coelho, L. S. Ochi, R. Haghnazar, D. Zuidema, M. S. Lima, and A. R. da Costa, "A multi-objective green uav routing problem," *Computers & Operations Research*, vol. 88, pp. 306–315, 2017.

[16] P. Shaw, "Using constraint programming and local search methods to solve vehicle routing problems," in *International Conference on Principles and Practice of Constraint Programming*. Springer, 1998, pp. 417–431.

[17] P. Venkatesh and A. Singh, "Two metaheuristic approaches for the multiple traveling salesperson problem," *Applied Soft Computing*, vol. 26, pp. 74–89, 2015.