Appendix

8.1. Proofs of theoretical results for the general case

In Section 4 we proved the results for the case that the drone is as least as fast as the truck between all pairs of nodes. In this section we will show how these results can be transferred to the general case where the drone is not necessarily faster than the truck but can also be slower between some or all pairs of nodes.

Consider a TSP-D tour $(\mathcal{R}, \mathcal{D})$. When the drone is slower than the truck on some edges, inequality (2) does not necessarily hold any more. We denote by K' the index set of the operations in which the drone leaves the truck and observe that

$$t(\mathcal{R}, \mathcal{D}) \geq \max\{c(\mathcal{R}), \sum_{k \in K'} c^d(\mathcal{D}_k)\}. \tag{24}$$

Similarly to Lemma 1 we can show the stronger inequality from Lemma 4

LEMMA 4. *Let $(\mathcal{R}, \mathcal{D})$ be a TSP-D tour and denote by K' the index set of the operations in which the drone leaves the truck. Then*

$$\sum_{k \in K'} c^d(\mathcal{D}_k) \geq 2 \cdot \sum_{v \in V_\mathcal{D} \setminus V_\mathcal{R}} \min_{w \in V_\mathcal{R}} c(v, w). \tag{25}$$

Proof Again, because in every operation there is at most one drone node, we know that the drone visits a combined node in $V_\mathcal{R} \cap V_\mathcal{D}$ right before and right after visiting a drone node in $V_\mathcal{D} \setminus V_\mathcal{R}$. Hence we have

$$\sum_{k \in K'} c^d(\mathcal{D}_k) = \sum_{e \in \mathcal{D} \setminus \mathcal{R}} c(e) \geq 2 \cdot \sum_{v \in V_\mathcal{D} \setminus V_\mathcal{R}} \min_{w \in V_\mathcal{R}} c(v, w). \tag{26}$$

□

Proof of Theorem 1 Let $(\mathcal{R}^*, \mathcal{D}^*)$ be an optimal solution to the TSP-D and $\mathcal{R}^{\text{TSP}}$ be an optimal solution to the TSP.

From $(\mathcal{R}^*, \mathcal{D}^*)$ construct a TSP tour $\mathcal{R}$ like in the proof for the special case in Section 4: Start with $\mathcal{R}^*$. For every drone node $v \in V_{\mathcal{D}^*}$, we pick the node $w \in V_{\mathcal{R}^*}$ which is closest to v with respect to the driving time c and insert arcs (w, v) and (v, w) into the tour and obtain

$$c(\mathcal{R}^{\text{TSP}}) \leq c(\mathcal{R}) = c(\mathcal{R}^*) + 2 \cdot \sum_{v \in V_{\mathcal{D}^*} \setminus V_{\mathcal{R}^*}} \min_{w \in V_{\mathcal{R}^*}} c(v, w).$$

We proceed analogously to the proof for the special case in Section 4: Using (25) we conclude

$$c(\mathcal{R}^{\text{TSP}}) \leq c(\mathcal{R}^*) + c(\mathcal{D}_d^*). \tag{27}$$

On the other hand, from (24) we obtain

$$t(\mathcal{R}^*, \mathcal{D}^*) \geq \max\{c(\mathcal{R}^*), \sum_{k \in K'} c^d(\mathcal{D}_k)\} \geq \max\{c(\mathcal{R}^*), \frac{1}{\alpha} \sum_{k \in K'} c^d(\mathcal{D}_k)\}, \tag{28}$$

where the last inequality follows from the definition of α. Finally combining (27) and (28)

$$t(\mathcal{R}^*, \mathcal{D}_d^*) \geq \max\{c(\mathcal{R}^{\text{TSP}}) - \sum_{k \in K'} c^d(\mathcal{D}_k), \frac{1}{\alpha} \sum_{k \in K'} c^d(\mathcal{D}_k)\} \geq \frac{c(\mathcal{R}^{\text{TSP}})}{1+\alpha}.$$

We conclude that

$$t(\mathcal{R}^{\text{TSP}}, \mathcal{R}^{\text{TSP}}) = c(\mathcal{R}^{\text{TSP}}) \leq (1+\alpha) t(\mathcal{R}^*, \mathcal{D}^*).$$

□

Proof of Lemma 2 for the general case We proceed analogously to the proof in Section 4: Based on the node sets $V_{\mathcal{R}^*}$ of $\mathcal{R}^*$ and $V_{\mathcal{D}^*}$ of $\mathcal{D}^*$ we construct a spanning tree T' of G as in the proof for the special case in Section 4: first, we remove one edge from $\mathcal{R}^*$ to obtain a spanning tree $T_{\mathcal{R}^*}$ of the node set $V_{\mathcal{R}^*}$. Then we connect all nodes v from the node set $V_{\mathcal{D}^*} \setminus V_{\mathcal{R}^*}$ to the closest node in $V_{\mathcal{R}^*}$. Hence

$$c(T') = c(T_{\mathcal{R}^*}) + \sum_{v \in V_{\mathcal{D}^*} \setminus V_{\mathcal{R}^*}} \min_{w \in V_{\mathcal{R}^*}} c(v, w). \tag{29}$$

Using (25), $c(T_{\mathcal{R}^*}) \leq c(\mathcal{R}^*)$, and (29) we obtain

$$\begin{aligned}
t(\mathcal{R}^*, \mathcal{D}^*) &\geq \max\{c(\mathcal{R}^*), \sum_{k \in K'} c^d(\mathcal{D}_k)\} && (30)\\
&= \max\{c(\mathcal{R}^*), \sum_{e \in E_{\mathcal{D}^*} \setminus E_{\mathcal{R}^*}} c^d(e)\} && (31)\\
&\geq \max\{c(\mathcal{R}^*), \frac{1}{\alpha} \sum_{e \in E_{\mathcal{D}^*} \setminus E_{\mathcal{R}^*}} c(e)\} && (32)\\
&\geq \max\{c(T_{\mathcal{R}^*}), \frac{2}{\alpha} \sum_{v \in V_{\mathcal{D}^*} \setminus V_{\mathcal{R}^*}} \min_{w \in V_{\mathcal{R}^*}} c(v, w)\} && (33)\\
&= \max\{c(T_{\mathcal{R}^*}), \frac{2}{\alpha}(c(T') - c(T_{\mathcal{R}^*}))\} && (34)\\
&\geq \frac{2}{2+\alpha} c(T') && (35)\\
&\geq \frac{2}{2+\alpha} c(T) && (36)
\end{aligned}$$

for the minimum spanning tree T of G. Hereby, inequality (35) holds because $\max_{x \in \mathbb{R}^+}\{x, \frac{2}{\alpha}(c(T') - x)\}$ is minimal if $x = \frac{2}{\alpha}(c(T') - x)$. □

Then Theorem 2 follows as detailed in the proof in Section 4.

It is easy to check that all results from Section 4 are still valid if non-zero recharging times and pickup and delivery times *of the drones* are considered, since these would not affect the time duration of the tour $(\mathcal{R}^{\text{TSP}}, \mathcal{R}^{\text{TSP}})$ but would slow down the duration of each TSP-D tour which uses the drone.

Speed-up for the exact partitioning algorithm from Section 6.2.2

To compute $T(i,j,k)$ in time $O(n^3)$ we proceed as follows. For each k compute

- $\text{SUM}(k-1,k+1,k) := c(r_{k-1},r_{k+1})$ and $T_k(k-1,k+1) = \max\{[c^d(r_{k-1},r_k) + c^d(r_k,r_{k+1})],\text{SUM}(k-1,k+1,k)\}$,
- for each $i < k-1$: $\text{SUM}(i,k+1,k) = c(r_i,r_{i+1}) + \text{SUM}(i+1,k+1,k)$ and $T(i,k+1,k) = \max\{[c^d(r_i,r_k) + c^d(r_k,r_{k+1})],\text{SUM}(i,k+1,k)\}$,
- for each $i < k-1$, $j > k+1$: $\text{SUM}(i,j,k) = c(r_{j-1},r_j) + \text{SUM}(i,j-1,k)$ and $T(i,j,k) = \max\{\alpha[c^d(r_i,r_k) + c^d(r_k,r_j)],\text{SUM}(i,j,k)\}$,

Hence we find $T(i,j,k)$ in computation time $O(n^3)$.

Furthermore, we can speed up our algorithm by not computing all $T(i,j,k)$, but only the ones for operations which can be part of an optimal solution: It is easy to see (from the triangle inequality) that if

$$\text{SUM}(i,j,k) \geq [c^d(r_i,r_k) + c^d(r_k,r_j)] \tag{37}$$

it follows that

$$\begin{aligned} T(i,j+1,k) =& \max\{[c^d(r_i,r_k) + c^d(r_k + r_j)],\text{SUM}(i,j,k) + c(r_i,r_j)] \\ =& \max\{[c^d(r_i,r_k) + c^d(r_k + r_j)],\text{SUM}(i,j,k)] + c(r_i,r_j) \\ =& T(k,i,j) + T(j,j+1,-1) \end{aligned}$$

and by induction the same holds for all $j' > j$. With a symmetric argument, we obtain the same results for all $i' < i$. Hence, as soon as for a fixed k and some i,j we have reached (37) we do not have to compute 'larger' operations and can move on to the next k.

8.2. The greedy algorithm: correctness

The greedy algorithm works on an ordered sequence of nodes $r_0,\ldots,r_{N+1}$ provided by the TSP tour $\mathcal{R}'$ computed in the first part of the heuristic, see Section 6.1 of the paper. The nodes are assigned one of the following labels: *simple*, *truck*, *drone* or *combined*. Initially all labels are set to *simple* and in each step of the greedy algorithm takes an action in which at least one *simple* label is updated to one of the other labels. The greedy algorithm stops when there are no simple labels left, or when there is no action that does not result in an infeasible solution or solution with infinite cost. As such, it is clear that N is an upper bound on the number of steps the greedy algorithm performs.

The solution for the TSP-D can be derived from the sequence of labels as follows. The tour of the truck is obtained by removing all locations which have a *drone* label from the sequence. Similarly, the tour of the drone is obtained by removing all locations which have a *truck* label

from the sequence. Based on this decomposition into two sub-tour, we can interpret the sequence of labels as a sequence of operations. If all nodes have a *simple* label, we have n operations. All nodes except the one at the boundary of the sequence are the end node of one operation and the start node of another.

In general, nodes with a *simple* or *combined* label decide at which points one operation ends and another one starts.

Not every sequence of labels will result in a feasible drone tour: two consecutive drone labels result in the drone covering two locations without returning to the truck. Because of this, the permitted actions are defined in such a way that the following invariant holds throughout the execution of the algorithm:

INVARIANT 1. For any node r_i with label *simple* or *combined*, the operation that starts at node r_i contains at most one node with a *drone* label.

In each step of the algorithm, the greedy algorithm considers one of the following procedures: *makeFly*, *pushLeft* and *pushRight* on a simple node r_i. In Algorithm 3 we give pseudo code for these procedures.

Procedure `makeFly`(r_i)
  if $r_i \neq$ *simple* $\vee$ $i \in \{0, N+1\}$ **then**
    throw exception: operation not possible
  end
  $r_i \leftarrow$ *drone* ;
  $r_{i-1} \leftarrow$ *combined* ;
  $r_{i+1} \leftarrow$ *combined* ;
Procedure `pushLeft`(r_i)
  if $r_i \neq$ *simple* $\vee$ $r_{i-1} \neq$ *combined* **then**
    throw exception: operation not possible
  end
  $r_i \leftarrow$ *combined* ;
  $r_{i-1} \leftarrow$ *truck* ;
Procedure `pushRight`(r_i)
  if $r_i \neq$ *simple* $\vee$ $r_{i+1} \neq$ *combined* **then**
    throw exception: operation not possible
  end
  $r_i \leftarrow$ *combined* ;
  $r_{i+1} \leftarrow$ *truck* ;

Algorithm 3: The makeFly, pushLeft and pushRight procedures

As long as there are remaining simple nodes, the greedy procedure looks for the best action and executes it, as is discussed in Section 6.2.1.

Although we have already established that this greedy procedure terminates in at most $O(N)$ iterations, we still need to establish that the resulting solution corresponds to a feasible truck-and-drone tour.

LEMMA 5. *A greedy procedure that executes feasible actions based on the procedures defined in Algorithm 3 maintains Invariant 1 during each iteration. As a result, the final sequence of labels corresponds to a feasible truck-and-drone tour.*

T o prove that the invariant holds in each operation, we use induction on the procedures and show that each of the procedures does not violate the invariant.

Base case: Initially all nodes have label *simple*. As there are no nodes with the label fly, the invariant holds.

Inductive case (makeFly): when we perform *makeFly* on a simple node r_i, we know that the resulting operation which starts at r_{i-1} and ends at r_{i+1} has precisely one node with a *drone* label and is thus consistent with the invariant. The induction hypothesis implies that the other operations are also consistent with the invariant.

Inductive case (pushLeft/pushRight): We do not introduce any new *drone* labels. As the induction hypothesis implies that the invariant held before the pushLeft or pushRight action, it still holds afterward.

We have now shown that the greedy algorithm introduced in Section 6.2.1 produces a feasible truck-and-drone tour. However, a naive implementation of the algorithm will consider all $O(N)$ potential actions during each operation. In the next section we discuss how this can be avoided an how we end up with an $O(N \log N)$ running time.

8.3. The greedy algorithm: obtaining $O(N \log N)$ running time

In order to achieve a running time of O(NlogN) for the greedy algorithm, we need two things. First, we need to consider how the savings for each possible action can be computed in $O(1)$ time and second we need to argue that the number of action for which the savings need to be updated after an action is performed is $O(1)$ as well.

DEFINITION 1. The savings of an action on node r_i can be expressed based on the operation o_j that ends at node r_i, the operation o_{j+1} that starts at node r_i and the new operation $o_{j'}$ that is create by the operation. The savings of this action are computed by $c(o_j) + c(o_{j+1}) - c(o_{j'})$.

LEMMA 6. *Performing either a* makeFly, pushLeft *or* pushRight *procedure will only change the savings of a constant number of actions.*

W hen an action is performed on a node r_i, we take an operation o_j that ends in r_i and the operation o_{j+1} that starts with r_i, and merge them into a new operation $o_{j'}$. As a result, the savings of the pushLeft and pushRight actions for the two nodes directly adjacent to the start and end nodes of the new operation $o_{j'}$ can be affected and need to be recomputed. All other actions will impact two operations other than $o_{j'}$ and as a consequence the savings of these actions are not impact by the current action.

Based on this lemma, we establish that we do not have to recompute all savings after we perform an action, but only a constant number of savings. We can exploit this fact with a priority queue. We use the priority queue to keep track of the actions in such a way that we can also obtain and remove the action with the lowest savings in $O(\log N)$ time, and after performing the action update the savings of a constant number of actions, each in $O(\log N)$ time as well.

Let us now consider how we can compute the new savings in $O(1)$ time after an action is executed. For this purpose we introduce eight arrays: *next*, *prev*, *flyNext*, *flyPrev*, *driveBefore*, *driveAfter*, *droneBefore* and *droneAfter*. The purpose of these arrays is to store additional information that is used to compute the savings of an action in constant time. The *next* and *prev* arrays hold indices of the next and previous nodes on the truck tour, while the *flyNext* and *flyPrev* arrays hold the next and previous bides on the drone path, in case the drone moves separately from the truck. If node r_i has a combined label, the array *driveBefore* holds the driving time of the truck within the operation that ends at node r_i at index i. Similarly, *driveAfter* holds the driving time of the operation that starts at node r_j at index j. Finally, *droneBefore* holds the flight time of the drone for the operation that ends in the node at the associated index, while *droneAfter* holds the flight time for operation that ends in the node at the associated index.

In order to keep the proper values in these arrays, we adapt the procedures in such a way that accurate information is always stored for the nodes that currently have a *simple* label or a *combined* label (but not necessarily for the other label types). The adapted version of the procedures is presented in Algorithm 4.

Using the information stored in these arrays, we can now introduce the pseudo code for the computation of the savings of each action in $O(1)$. This pseudo code is presented in Algorithm 5.

We now have the necessary elements to analyze the running time of the full algorithm. On a high level, the greedy algorithm is defined by the pseudo code in Algorithm 6.

Based on the observations thus far, we can finally establish the running time of the greedy algorithm.

THEOREM 3. *The improved greedy algorithm runs in $O(N \log N)$ time.*

Procedure `Initialize()`
  for $i \leftarrow 0, \ldots, N+1$ **do**
    next[i] $\leftarrow i+1$;
    prev[i] $\leftarrow i-1$;
    flyNext[i] $\leftarrow$ undefined ; flyPrev[i] $\leftarrow$ undefined ;
    driveBefore[i] $\leftarrow 0$; driveAfter[i] $\leftarrow 0$;
    flyBefore[i] $\leftarrow 0$; flyAfter[i] $\leftarrow 0$;
  end
Procedure `makeFly(`r_i`)`
  if $r_i \neq$ *simple* $\vee$ $i \in \{0, N+1\}$ **then**
    throw exception: operation not possible
  end
  $r_i \leftarrow$ *drone* ;
  $r_{i-1} \leftarrow$ *combined* ;
  $r_{i+1} \leftarrow$ *combined* ;
  next$[i-1] \leftarrow i+1$; prev$[i+1] \leftarrow i-1$;
  flyNext$[i-1] \leftarrow i$; flyNext$[i] \leftarrow i+1$;
  flyPrev$[i] \leftarrow i-1$; flyPrev$[i+1] \leftarrow i$;
  driveBefore$[i+1]$, driveAfter$[i-1] \leftarrow c(v_{i-1}, v_{i+1})$;
  flyBefore$[i+1]$, flyAfter$[i-1] \leftarrow c^d(v_{i-1}, v_i) + c^d(v_i, v_{i+1})$;
Procedure `pushLeft(`r_i`)`
  if $r_i \neq$ *simple* $\vee$ $r_{i-1} \neq$ *combined* **then**
    throw exception: operation not possible
  end
  $r_i \leftarrow$ *combined* ;
  $r_{i-1} \leftarrow$ *truck* ;
  $j \leftarrow$ flyPrev$[i-1]$; $k \leftarrow$ flyPrev$[j]$;
  flyPrev$[i] \leftarrow j$; flyNext$[j] \leftarrow i$;
  driveBefore$[i] \leftarrow$ driveBefore$[i-1] + c(v_{i-1}, v_i)$;
  driveAfter$[k] \leftarrow$ driveBefore$[i]$;
  flyBefore$[i]$, flyAfter$[k] \leftarrow c^d(v_k, v_j) + c^d(v_j, v_i)$;
Procedure `pushRight(`r_i`)`
  if $r_i \neq$ *simple* $\vee$ $r_{i+1} \neq$ *combined* **then**
    throw exception: operation not possible
  end
  $r_i \leftarrow$ *combined* ;
  $r_{i+1} \leftarrow$ *truck* ;
  $j \leftarrow$ flyNext$[i+1]$; $k \leftarrow$ flyNext$[j]$;
  flyNext$[i] \leftarrow j$; flyPrev$[j] \leftarrow i$;
  driveAfter$[i] \leftarrow$ driveAfter$[i+1] + c(v_i, v_{i+1})$;
  driveBefore$[k] \leftarrow$ driveAfter$[i]$;
  flyAfter$[i]$, flyBefore$[k] \leftarrow c^d(v_i, v_j) + c^d(v_j, v_k)$;

Algorithm 4: Adapted procedures for makeFly, pushLeft and pushRight.

Procedure `makeFlySavings` (r_i)
  if $r_i \neq$ *simple* $\vee\ i \in \{0, N+1\}$ **then**
    return $-\infty$
  end
  oldCost $\leftarrow c(v_{i-1}, v_i) + c(v_i, v_{i+1})$;
  newCost $\leftarrow \max\left\{c(v_{i-1}, v_{i+1}), c^d(v_{i-1}, v_i) + c^d(v_i, v_{i+1})\right\}$;
  return *oldCost* − *newCost*
Procedure `pushLeftSavings` (r_i)
  if $r_i \neq$ *simple* $\vee\ r_{i-1} \neq$ *combined* **then**
    return $-\infty$
  end
  oldCost $\leftarrow \max\{$ driveBefore$[i-1]$, flyBefore$[i-1]$ $\} + c(v_{i-1}, v_i)$;
  $j \leftarrow$ flyPrev$[i-1]$; $k \leftarrow$ flyPrev$[j]$;
  newDrive $\leftarrow$ driveBefore$[i-1]$ $+c(v_{i-1}, v_i)$;
  newFly $\leftarrow c^d(v_k, v_j) + c^d(v_j, v_i)$;
  return *oldCost* $-\max\{$ *newDrive* , *newFly* $\}$
Procedure `pushRightSavings` (r_i)
  if $r_i \neq$ *simple* $\vee\ r_{i+1} \neq$ *combined* **then**
    return $-\infty$
  end
  oldCost $\leftarrow \max\{$ driveAfter$[i+1]$, flyAfter$[i+1]$ $\} + c(v_i, v_{i+1})$;
  $j \leftarrow$ flyNext$[i+1]$; $k \leftarrow$ flyNext$[j]$;
  newDrive $\leftarrow$ driveAfter$[i+1]$ $+ c(v_i, v_{i+1})$;
  newFly $\leftarrow c^d(v_i, v_j) + c^d(v_j, v_k)$;
  return *oldCost* $-\max\{$ *newDrive* , *newFly* $\}$

Algorithm 5: Procedures that compute the savings of each action in constant time, based on the information that is stored in a number of arrays.

Initialize: Priority queue q with the savings of all actions
while *actions left in q with savings* $> -\infty$ **do**
  Pop action a with maximum savings from the queue ;
  Execute action a ;
  Update the savings of the affected actions in the queue ;
end
Compute a tour based on the labels ;

Algorithm 6: Description of the greedy algorithm with a priority queue

We know that the number of potential actions at any time is $O(N)$, so the length of the priority queue is also $O(N)$ at any time. Pop and update operations can be performed in $O(\log N)$

time on a priority queue. Based on Lemma 6 we know that the savings of only a constant number of actions are affected after an action is performed. Furthermore, the action itself can be performed in constant time. This implies that the first line of the loop body takes $O(\log N)$ to execute, the second line can be executed in constant time, and updating a constant number of savings in the third line is again $O(\log N)$. As we had established before that the greedy algorithm can do at most $O(N)$ iterations, this results in a running time of $O(N \log N)$.