33RD CONGRESS
OF THE INTERNATIONAL COUNCIL
OF THE AERONAUTICAL SCIENCES
STOCKHOLM, SWEDEN, 4-9 SEPTEMBER, 2022

ICAS 2022 SWEDEN

AN ENGINELESS TAXI OPERATIONS SYSTEM USING BATTERY-OPERATED AUTONOMOUS TOW TRUCKS

Ing. Stefano Zaninotto[1], Dr Ing. Jason Gauci[2] & Dr Ing. Brian Zammit[3]

[1]Research Support Officer II, Institute of Aerospace Technologies, University of Malta, Malta
[2]Senior Lecturer, Institute of Aerospace Technologies, University of Malta, Malta
[3]Senior Lecturer, Electronic Systems Engineering, Faculty of Engineering, University of Malta, Malta

Abstract

One of the solutions proposed by the aerospace industry to reduce fuel consumption, air pollution and noise at airports is to introduce electric trucks to tow aircraft from the stand to the runway (or vice-versa). However, the introduction of tow trucks increases surface traffic which, from an Air Traffic Controller's (ATCo) point of view, is undesirable. Many solutions have been proposed to mitigate this increase in workload through the introduction of automated planning and execution. However, most of these solutions suffer from one or more of the following: severe limitations in the size of their solution spaces; inability to schedule and plan routes for multiple active runways simultaneously; and inability to consider battery state-of-charge when assigning tow trucks. In terms of performance testing of such solutions, only singular performance metrics (e.g., number of potential conflicts between vehicles or average taxi time) have been considered in the literature, thus limiting the validity and applicability of the results. This paper details a novel system for taxi operations using autonomous tow trucks in order to improve ground operations and overcome some of the limitations of existing solutions. The system identifies conflict-free solutions that minimise taxi-related delays and taxi route lengths, while maximising the use of the tow trucks for taxi operations. It can cater for multiple active runways and accounts for tow truck battery state-of-charge, as well as limits in the number of tow trucks and charging stations. The proposed algorithm was tested for a large number of scenarios and was evaluated using various performance metrics. The results show that the algorithm is capable of utilising the tow trucks for aircraft taxiing without creating any traffic conflicts. To achieve this, almost 70% of the flights had to be slightly delayed to ensure adequate traffic separation is maintained at all times, with delays of up to 3 minutes. Furthermore, the algorithm was capable of assigning tow trucks to more than 80% of the flights, even with a tow truck fleet as small as 25% of the airfield hourly traffic.

Keywords: engineless taxiing, tow trucks, Dijkstra

1. Introduction

The growth of air traffic during the last decades has significantly impacted the environment in terms of fuel emissions, air and noise pollution. These effects have been recognized by the European Commission and strict targets have been identified for the future through initiatives such as "Flight Path 2050" [1] and the European Green Deal [2]. The first document sets ambitious targets, such as a 70% reduction in CO2, a 90% reduction in NOx, and a reduction in noise emissions in comparison to levels in the year 2000; while the second document targets emission reductions of 55% by 2030 in comparison to levels in the year 2019. In addition to these requirements, all taxiing procedures will be required to be carbon neutral by 2050.

Historically, efforts related to reducing emissions have mainly focused on the airborne phase of flight as this constitutes the majority of the flight's duration. In addition, since the aircraft's engines are optimised for cruise operations, they are highly inefficient when taxiing. Furthermore, high traffic levels at airports, coupled with inefficiencies in the management of taxi operations, often lead to congestion on taxiways and queues at runway holding points. Such situations necessitate repeated stop-and-go movements and introduce long engine idle times which increase emissions levels. Unsurprisingly, aircraft are considered to be the largest single emission source at airports [3].

Reducing emissions throughout the taxi phase of flight is one of the challenges that is being addressed by the Single European Sky ATM Research Joint Undertaking (SESAR JU) programme and two main technologies are being considered by the aerospace industry [4]. One of these technologies is that adopted by systems such as WheelTug [5] and the Electric Green Taxiing System (EGTS) [6], which relies on the use of electric motors installed in the main (or nose) landing gear of the aircraft. An alternative technology relies on the use of manned or unmanned electric tow trucks to tow aircraft from the stand to the runway (or vice-versa). For instance, the TaxiBot solution [7] uses a semi-robotic, pilot-controlled tow truck which has been successfully tested and is currently in use at Frankfurt Airport. This solution does not require modifications to the aircraft and does not add to aircraft weight. However, the introduction of tow trucks to taxi operations increases surface traffic at the airport, potentially increasing ATCo workload and creating congestion. This work focuses on the latter technology.

A number of concepts and systems which make use of tow trucks for taxi operations have been proposed in the literature and were reviewed in previous work [8]. In [8], Zaninotto et al. proposed an algorithm capable of overcoming some of the limitations of these concepts and systems; however, this algorithm is only partially autonomous as it requires the intervention of ATCos to issue clearances. Furthermore, it does not always identify conflict-free routes and was designed specifically for one particular location (Malta International Airport), which limits the applicability of the solution. Other limitations of the algorithm include its inability to adapt to a variable number of tow trucks, and its inability to take into account the state-of-charge of tow truck batteries during scheduling.

This paper proposes a re-design of the system in [8] which aims to fully automate and optimise the taxi phase by assigning tow trucks to aircraft and generating conflict-free routes to satisfy at pre-defined flight schedule. The proposed algorithm is tested in a simulation environment that was developed for the testing of engine-less taxi solutions [9] and performance analysis is carried out by defining a number of performance metrics and extracting statistical data from a large number of scenarios.

The rest of the paper is organised as follows. Section 2 describes the design of the algorithm. Section 3 defines the performance metrics and test scenarios, and presents and discusses the results. Finally, Section 4 outlines the key conclusions of this paper and highlights areas for future work.

2. Design of Algorithm

The algorithm described in this section is partly based on the previous work [8], where various elements of the aerodrome environment in the context of taxi operations were designed and developed. These include the modelling of the airport layout (through the use of a directed graph), the generation of flight schedules, and the modelling of aircraft and tow truck movement. The core functionality has been redesigned based on the principle that ATCo intervention is now not required in any part of the process. This means that the system is fully autonomous, thus imposing no additional ATCo workload.

The core algorithm was implemented in Matlab and a control flow graph of the system is shown in Figure 1. The algorithm consists of nine main modules that will be detailed in the following sections.

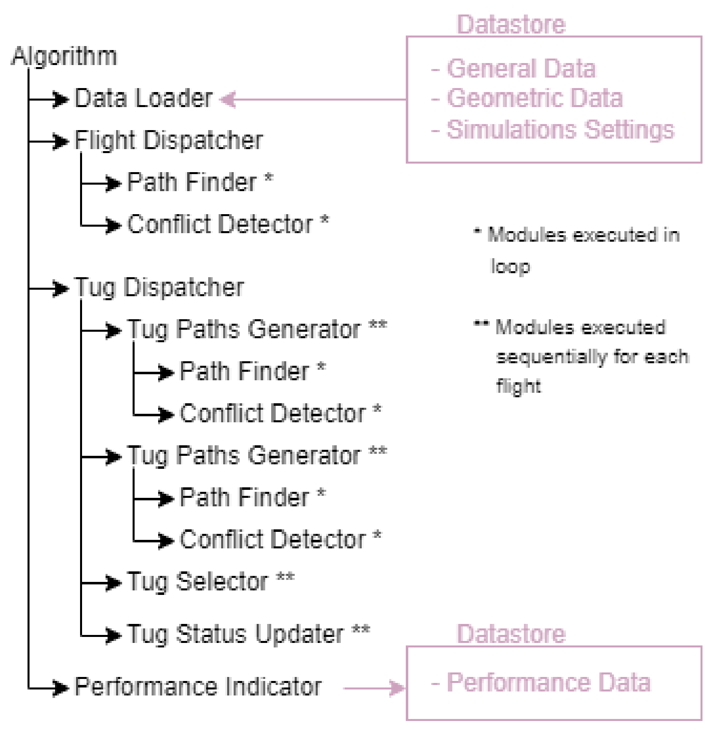

Figure 1 - Control flow graph of the modules of the system.

2.1 Data Loader and Data Store

From the *Data Store*, the *Data Loader* extracts layout information related to the selected airport, relevant simulation settings (e.g., number of flights and number of tow trucks per depot), and creates the airport environment together with a random flight schedule. The *Data Loader* and the *Data Store* were extensively described in previous work [9].

2.2 Flight Dispatcher

The *Flight Dispatcher* is responsible for identifying conflict-free routes between the assigned runway and the allocated stand for arrivals, or between the stand and the runway for departures.

The algorithm attempts to minimise the delay of each flight, which is indicated by the Solution Cost (*SC*) and is given by:

$$SC = TTD + STD \tag{1}$$

where *TTD* is Taxi Time Delay and *STD* is the Start Time Delay.

The *TTD* is given by:

$$TTD = ATT - ITT \tag{2}$$

where *ATT* is the Actual Taxi Time on the assigned path and *ITT* is the Ideal Taxi Time, measured along the ideal (shortest) taxi path.

STD is the delay accumulated by the aircraft while waiting next to the runway (for arrivals) or at the stand (for departures) and is given by:

$$STD = AST - FST \tag{3}$$

Where *AST* is the Actual Start Time when the aircraft starts taxiing and *FST* is the Flight Scheduled Time.

The pseudocode for this module is shown in Figure 2.

```
for flight = 1, number of flights
    iteration =1
    for runway = 1, number of runways
        runway iteration = 0
        STD = 0
        runway node = find node (runway)
        parking node = find node (parking)
        if flight is arrival
            $n_{st}$ = runway node
            $n_{end}$ = parking node
        else if flight is departure
            $n_{st}$ = parking node
            $n_{end}$ = runway node
        end
        run Path Finder
            (inputs: nodes, edges, $n_{st}$, $n_{end}$)
            (outputs: ideal taxiing distance)
        ITT = ideal taxiing distance / $v_a$
        while runway iteration $\leq$ runway iterations cap
            iteration = iteration + 1
            runway iteration = runway iteration + 1
            AST = FST + STD
            set conflict free path as false
            set ECL as empty
            while conflict free path is false
                run Path Finder
                    (inputs: nodes, edges, $n_{st}$, $n_{end}$, ECL)
                    (outputs: $P_{nodes}$, taxiing distance, path feasibility)
                if path feasibility is false
                    set SC as infinite
                    break loop
                else if path feasibility is true
                    ATT = taxiing distance / $v_a$
                    TTD = ATT – ITT
                    $P_{times}$ = AST + times (path nodes) / $v_a$
                    run Conflict Detector
                        (inputs: edges, ECL, $P_{nodes}$, $P_{times}$, GOT)
                        (outputs: ECL, conflict free path, VOT)
                end
            end
            SC = STD + TTD
            save [$P_{nodes}$, $P_{times}$, STD, conflict free path, SC, VOT] as solution (iteration)
            if STD (iteration) $\leq$ any (SC (1, …, iteration))
                STD = STD + delay increment
            else
                break loop
            end
        end
    end
    if all (SC (1, …, iteration) is infinite)
        set simulation feasibility false
        break loop
    end
    selected iteration = find iteration with minimum (SC (1, …, iteration)))
    selected solution = solution (selected iteration)
    GOT = append (VOT (selected iteration)) to GOT
end
```

Figure 2 - Pseudocode for the *Flight Dispatcher* module.

Each flight is analysed sequentially and solutions are explored for each active runway, as follows.

First, as shown in lines 8-18 of the pseudocode in Figure 2, the path's start node (n_{st}), equal to the runway node for an arrival and to the parking node for a departure, and the path's end node (n_{end}), equal to the parking node for an arrival and to the runway node for a departure, are set. These nodes are input to the *Path Finder* module (see Section 2.3), in which the ideal (i.e. shortest) taxi distance is computed. This distance is then divided by an average speed (v_a), set equal to 10 *m/s*, to find the ideal taxi time.

Then, the module attempts to find a conflict-free solution. First, the *Path Finder* is executed once again (lines 26-28 of the pseudocode in Figure 2) and a path, consisting of a Set of Path Nodes (P_{nodes}), is generated; then, a Set of Path Times (P_{times}), indicating when the aircraft will traverse each node, is calculated (as explained in Section 2.3) and the feasibility of the path – indicating whether the module found a feasible path – is checked.

If the path is not feasible, the solution is discarded; otherwise P_{nodes} and P_{times} are processed by the *Conflict Detector* (described in Section 2.4). This module checks if the path is conflict-free; produces a Vehicle Occupation Table (VOT), which stores all the time windows during which the edges of the path are occupied by the vehicle; and, if potential conflicts are detected, stores them in the Edges in Conflict List (ECL) (lines 29-38 of the pseudocode in Figure 2).

This process (indicated in lines 25-40 of the pseudocode in Figure 2) is repeated until a conflict-free path is identified or, as mentioned, until the path is flagged as not feasible. In case a conflict-free path is found, the module calculates and store the *SC* of the current iteration (lines 41-42 of the pseudocode in Figure 2).

Then, a new iteration is set, and the *STD* is incremented by 10 *s*. New solutions are calculated until the *STD* of a new solution is greater than or equal to the *SC* of any solution, making useless the search for further solutions for the analysed active runway. The whole process (lines 3-49 of the pseudocode in Figure 2) is repeated for the following runway until all the active runways are analysed.

In case no conflict-free solutions are found (meaning that all of the solutions are discarded, as all of the corresponding paths are considered to be unfeasible), the simulation is marked as unfeasible and the algorithm stops the calculations for the selected simulation (lines 50-53 of the pseudocode in Figure 2).

Finally, as shown in lines 54-56 of the pseudocode in Figure 2, the solution with the lowest *SC* is selected and the VOT of the selected solution is appended to the Global Occupation Table (GOT), which represents the combination of all the VOTs of the selected solutions of the previously analysed flights (therefore, when the first flight is analysed, the GOT is empty).

2.3 Path Finder

The *Path Finder* module computes the shortest path between two nodes using Dijkstra's algorithm. First, the module checks the ECL and excludes any edges contained in this list from the airport graph. Then, the shortest path between the n_{st} and n_{end} is determined. If the exclusion of certain edges results in an unfeasible path, the *Path Finder* indicates that that path is not feasible. Instead, if a feasible path is found, the function returns P_{nodes} and P_{times}. P_{nodes} is given by:

$$P_{nodes} = \{n_1, \dots, n_n, \dots, n_N\} \tag{4}$$

where:
n_1 is the first node of the path and is it equal to n_{st},
n_n is the nth node of the path,

n_N is the last node of the path and it is equal to n_{end}, and
N is the total number of nodes of the path.

P_{times}, which is calculated immediately after the execution of the *Path Finder* module, is given by:

$$P_{times} = \{t_1, \dots, t_n, \dots, t_N\} \tag{5}$$

where:
t_1 is the time when the vehicle passes through the first node n_1 and it is equal to *AST*,
t_n is the time when the vehicle passes through the nth node n_n, and
t_N is the time when the vehicle passes though the last node n_N.

2.4 Conflict Detector

The purpose of the *Conflict Detector* is to create a VOT; to determine if the path provided by the *Path Finder* is conflict-free; and to store the edges where potential conflicts are detected in the ECL. First, the Set of Path Edges (P_{edges}) of the path is determined from P_{nodes} as follows:

$$P_{edges} = \{e_1, \dots, e_n, \dots, e_E\} \tag{6}$$

where:
e_1 is the first edge of the path passing between nodes n_1 and n_2,
e_n is the nth edge of the path passing between nodes n_n and n_{n+1},
e_E is the last edge of the path passing between nodes n_{N-1} and n_N, and
E is the total number of edges of the path and it is equal to $N - 1$.

Then, the Set of Minimum Path Times ($P_{times.min}$) is calculated from P_{times}. $P_{times.min}$ represents the time when the vehicle enters each edge as follows:

$$P_{times.min} = \{tmin_1, \dots, tmin_n, \dots, tmin_N\} \tag{7}$$

where:
$tmin_1$ is the time when the vehicle enters the first edge, and is equal to $t_1 - t_b$,
$tmin_n$ is the time when the vehicle enters the nth edge, and is equal to $t_n - t_b$,
$tmin_E$ is the time when the vehicle enters the last edge, and is equal to $t_{N-1} - t_b$, and
t_b represents a buffer time equal to 10 *s*.

Finally, the Set of Maximum Path Times ($P_{times.max}$) is calculated using P_{times}. $P_{times.max}$ represents the time when the vehicle leaves each edge, as follows:

$$P_{times.max} = \{tmax_1, \dots, tmax_n, \dots, tmax_N\} \tag{8}$$

where:
$tmax_1$ is the time when the vehicle leaves the first edge, and it is equal to $t_2 + t_b$,
$tmax_n$ is the time when the vehicle leaves the n-edge, and it is equal to $t_{n+1} + t_b$, and
$tmax_E$ is the time when the vehicle leaves the last edge, and it is equal to $t_N + t_b$.

The purpose of the buffer time t_b is to slightly increase the time reserved for a vehicle to cross an edge. This is expected to increase the traffic separation and, in turn, the robustness of the system to uncertainties in vehicle positions and other parameters.

Next, $P_{times.min}$ and $P_{times.max}$ are compared with a table called the Time Windows Table. This table divides the simulation time into intervals (time windows) and assigns an ID to each of them. Table 1 shows an example of a Time Windows Table, with tw_{ID} representing the time window ID, $tw_{time.st}$ indicating the start time of each time window and $tw_{time.end}$ indicating the end time of each time window.

Table 1 - Time Windows Table.

tw_{ID}	$tw_{time.st}$	$tw_{time.end}$
1	00:00	00:10
2	00:10	00:20
3	00:20	00:30
4	00:30	00:40
5	00:40	00:50
6	00:50	01:00
…	…	…

For each edge in the path of a vehicle, the *Conflict Detector* identifies all the time windows which partially or completely overlap the interval between $P_{times.min}$ and $P_{times.max}$ values corresponding to the edge. All the identified pairs $\{e, tw_{ID}\}$ are then stored in the VOT.

In order to better understand how the *Conflict Detector* works, a simple graph is shown in Figure 3. This graph shows node IDs, edge IDs and a path (marked in red) between a START node and an END node. The time when the vehicle passes through each node is indicated in brackets. As explained in Section 2.3 and in this section, the first step is to obtain P_{nodes}, P_{times} and P_{edges}; then, the sets $P_{times.min}$ and $P_{times.max}$ are calculated (Table 2). For each edge e_n, the interval between $tmin_n$ and $tmax_n$ is then compared with the time intervals defined in the Time Windows Table (Table 1). The overlapping time windows are stored in the VOT, as shown in Table 3.

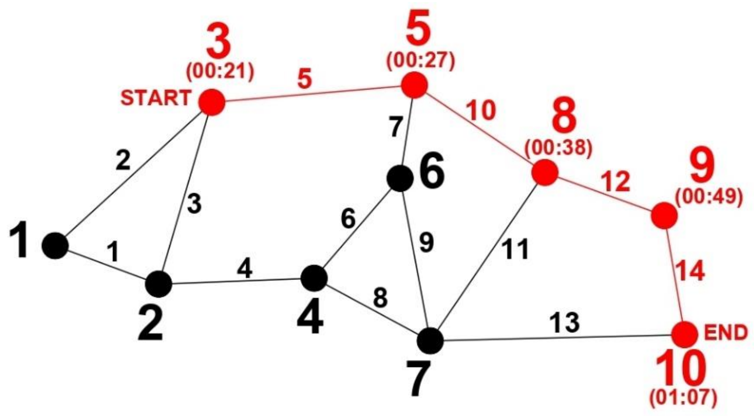

Figure 3 - Example of graph with path.

Table 2 - Path parameters in the *Conflict Detector* module.

P_{nodes}	P_{times}	P_{edges}	$P_{times.min}$	$P_{times.max}$
3	00:21	5	00:11	00:37
5	00:27	10	00:17	00:48
8	00:38	12	00:28	00:59
9	00:49	14	00:39	01:17
10	01:07	-	-	-

Finally, the *Conflict Detector* compares the VOT with the GOT (which includes the VOTs of all the vehicles analysed previously) and looks for pairs of edges and time windows which appear in both tables. If no matches are found – meaning that the analysed vehicle does not occupy any of the edges of previously analysed vehicles at the same time – the module considers the path to be conflict-free. Otherwise, the path is considered to conflict with other paths, and the edges contained in the matching pairs are added to the ECL (and, as explained in Section 2.3, they will be excluded from the airport graph by the *Path Finder* module when searching for a new shortest path). For instance, if both the VOT and GOT contain the pairs [10, 2], [10, 3] and [12, 5], Edges 10 and 12 will be included in the ECL.

Table 3 - Example of VOT.

e	tw_{ID}
5	2
5	3
5	4
10	2
10	3
10	4
10	5
12	3
12	4
12	5
12	6
14	4
14	5
14	6
14	7
14	8

2.5 Tug Dispatcher

If the *Flight Dispatcher* module is able to assign a route to each flight, the *Tug Dispatcher* is activated. This module is responsible for: assigning a tow truck to each flight; finding a conflict-free route for each tow truck between each depot and the start node of the aircraft, and between the end node of the aircraft and each depot; assigning a depot to each tow truck once a towing mission is completed; and updating the status of the assigned tow trucks and of their destination depots.

In order to accomplish this, the *Tug Dispatcher* analyses each flight sequentially and, for each flight, uses the sub-modules *Tug Paths Generator* in order to generate conflict-free routes (refer to Section 2.6); *Tug Selector* to assign a tow truck to the flight (refer to Section 2.7); and *Tug Status Updater* to update the status of the assigned tow trucks and of their destination depot (refer to Section 2.8).

The objectives of this module are: to maximise the use of the tow trucks available in the simulation (thus minimising the number of aircraft which need to taxi using their main engines); to generate routes for the tow trucks which do not conflict with any aircraft route assigned by the *Flight Dispatcher*; and to balance the load between the available tow trucks, allowing them to recharge in the depots when their battery level drops below a predefined threshold.

An important aspect of the *Tug Dispatcher* is that only tow trucks which are parked at a depot can be assigned to an aircraft. Therefore, after completing a mission, a tow truck must return to a depot before being assigned to a new mission.

2.6 Tug Paths Generator

The aim of the *Tug Paths Generator* is to generate conflict-free paths (each one with an *ATT*) which the tow trucks can follow to reach an aircraft on time, or to return to a depot. To accomplish this, the module is run twice: first to search for paths from all the depots to the attachment node (i.e. the aircraft start node) with the analysed flight, and another time from the detachment point (i.e. the aircraft end node) with the analysed flight to the depots. The *Tug Dispatcher* reports to the *Tug Paths Generator* the type of mission which has to be analysed – Reaching the Aircraft Mission Type (RAMT) or Returning to Depot Mission Type (RDMT) – and the module adapts the settings accordingly.

The operation of the *Tug Paths Generator* is almost identical to that of the *Flight Dispatcher*; however, there are a number of differences which should be taken into account. First, it is assumed that two unloaded tow trucks can pass through an edge (e.g. a taxiway) side by side. Therefore, if a conflict-free path is found by the Tug Paths Generator, the VOT is not added to the GOT, unlike what happens in the case of the *Flight Dispatcher*. This allows multiple tow trucks to use the same paths (or parts of them) at the same time.

While the *Flight Dispatcher* attempts to find solutions between the active runways and the designated gate (in the case of an arrival), or between the gate and the active runways (in the case of a departure), the *Tug Paths Generator* looks for solutions between the depots and the start node of the aircraft (in case of RAMT), or between the end node of the aircraft and the depots (in case of RDMT). Also, while the *Flight Dispatcher* selects only one solution, the *Tug Paths Generator* selects one solution for each depot, in order to provide a potential conflict-free route for each tow truck to reach the aircraft.

With regards to the RAMT, P_{times} is calculated in reverse, from the last node (equal to the aircraft start node) to the first node (equal to the depot). This is done because the tow trucks must arrive at the attachment point not later than the predefined attachment time (which is equal to the *AST* of the aircraft minus T_{attach}, which is the time taken for the tow truck to attach to an aircraft, and is assumed to be 30 *s*) in order not to disrupt the aircraft's route defined by the *Flight Dispatcher*. In the case of a departure, since the aircraft is waiting at a stand, the *Tug Paths Generator* explores more solutions, such as sending the tow truck to the stand earlier and comparing the *SCs* (as the *Flight Dispatcher* does). However, in the case of an arrival, this operation is not allowed in order to prevent the tow truck from waiting near the runway, as this might create traffic congestion.

With regards to the RDMT, P_{times} is calculated from the first node (equal to the detachment node) to the last node (equal to the depot), with t_1 equal to the detachment time, which is equal to the arriving time of the aircraft (calculated as the sum of *AST* and *ATT*) plus T_{detach} (which is the time taken for the tow truck to detach from an aircraft, and is assumed to be 30 *s*). In the case of an arrival, since the aircraft arrives at a stand, the *Tug Paths Generator* explores more solutions, such as postponing the departure of the tow truck from the stand and comparing the *SCs* (as the *Flight Dispatcher* does). However, in the case of a departure, this operation is not allowed in order to prevent the tow truck from waiting next to the runway, as this might create traffic congestion.

If, for both RAMT and RDMT, no solutions are found for any depot, the analysed aircraft cannot be towed – meaning that it will have to taxi using its main engines – and the *Tug Dispatcher* moves onto the next flight in the schedule.

2.7 Tug Selector

The *Tug Selector* is responsible for assigning a tow truck to an aircraft and identifying the depot which the tow truck returns to after the taxi mission is completed. In order to do this, the module analyses a number of tow truck parameters and the availability of each depot. These parameters are stored in two sets of tables: a set consisting of a Tug Status Table (TST) for each tow truck, which shows the status of the tow truck at each time window (see example in Table 4), and another set consisting of a Depot Status Table (DST) for each depot, which shows the status of the depot at each time window (see example in Table 5).

Table 4 - Example of part of a TST.

tw_{ID}	Depot ID	Parked	Loaded	Battery charge	Battery sufficient	Available
…	…	…	…	…	…	…
77	-	FALSE	FALSE	20.5%	TRUE	FALSE
78	-	FALSE	FALSE	20.4%	TRUE	FALSE
79	-	FALSE	TRUE	20.3%	TRUE	FALSE
80	-	FALSE	TRUE	20.1%	TRUE	FALSE
…	…	…	…	…	…	…

Table 5 - Example of part of a DST.

tw_{ID}	Number of available tugs	Number of avilable parking spots
…	…	…
77	1	5
78	1	5
79	1	5
80	1	5
…	…	…

For each time window, the TST indicates the depot in which the tow truck is located; whether the tow truck is parked; whether it is attached to an aircraft; its battery level; whether sufficient charge is available to handle a towing mission (the minimum threshold is set to 20%); and the tow truck's availability. A tow truck is only considered to be available if it is parked and adequately charged. The tow truck is assumed to have an endurance of 30 minutes with a full battery and an average load. With an average speed of v_a = 10 m/s, it is therefore able to travel a distance of 18 km on a single battery charge.

The DST indicates the availability of a depot for each time window in terms of the number of available tow trucks and parking spaces (charging points). The number of parking spaces available per depot is a design choice that is driven by the infrastructure of the airfield. To ensure an adequate number of free charging points, the number of parking spaces at each depot is defined as follows:

$$n_{parking.slots} = \lceil 1.5 \times (n_{tugs}/n_{depots}) \rceil \tag{9}$$

where:
n_{tugs} is the total number of tow trucks, and
n_{depots} is the number of depots in the airfield.

Using this equation, the total number of charging points is always greater than the number of tugs and scales in proportion to the size of the tow truck fleet.

The *Tug Selector* first excludes the parked tow trucks for which no conflict-free path to the analysed aircraft was found, as well as tow trucks that are unavailable between the relevant departure time window during which they should leave their depot ($tw_{ID.tug.st}$), as calculated by the *Tug Paths Generator*, and the time window during which they should detach from the aircraft ($tw_{ID.det}$). The list of remaining (non-excluded) tow trucks is the Candidate Tugs List (CTL).

Then, a destination depot is assigned to each of the tow trucks in the CTL based on three criteria:

1. the *Tug Paths Generator* successfully identifies a conflict-free route from the aircraft end node (i.e. detachment node) to the depot under consideration;

2. the depot under consideration has at least one available parking slot at the time window during which the tow truck arrives at the depot (*$tw_{ID.tug.end}$*); and

3. the cost (*SC*) of returning to the depot under consideration is less than the cost of returning to any other depot.

For each tow truck *i,* a towing cost, (c^i_{towing}) is defined as the total taxi mission time. In addition, the associated battery charge level, b^i, is imported from the TST. The battery charge level is converted into a battery cost, c^i_{bat}, and scaled to the magnitude levels of the towing cost using Equation (10):

$$c^i_{bat} = c_{towing.max} - \frac{c_{towing.max}}{(b_{max} - b_{min})} \times \left(b^i - b_{min}\right) \tag{10}$$

where:
$c_{towing.max}$ is the maximum c_{towing} cost of the tow trucks in the CTL,
b_{min} is the minimum battery level of the tow trucks in the CTL, and
b_{max} is the maximum battery charge set to 100%.

Equation (10) was designed with the aim of making it possible to add c^i_{bat} to c^i_{towing} (without restricting the maximum value of c^i_{bat} to 100%) and of assigning a low cost for a high battery charge (so as to promote tow trucks with higher levels of battery). In fact, Equation (10) results in zero battery cost for fully charged tow trucks. For partial charges, the cost is linearly increased up to a maximum of $c_{towing.max}$. Finally, the total cost of the towing mission c^i_{tot} is computed as follows:

$$c^i_{tot} = c^i_{towing} + c^i_{bat} \tag{11}$$

The tow truck with the minimum c^i_{tot} cost is selected and assigned to the aircraft under consideration.

2.8 Tug Status Updater

Once a tow truck is assigned to an aircraft, its Tug Status Table (TST) is updated from $tw_{ID.tug.st}$ to $tw_{ID.tug.end}$ (except for the battery charged which is updated until the last time window of the simulation). The battery charge is updated on the basis of the following three parameters:

- a discharging rate of 2*% per minute*, applied when the tow truck is in motion and unloaded;
- a discharging rate of 3*% per minute*, applied when the tow truck is in motion and loaded; and
- a charging rate of 2*% per minute*, applied when the tow truck is at a charging point in a depot.

Subsequently, all the DSTs are updated according to the new values of the TSTs.

Then, the *Tug Dispatcher* moves onto next flight in the schedule and repeats the whole process, as described in Sections 2.5 to 2.8.

2.9 Performance Indicator

At the end of each simulation, the *Performance Indicator* computes various metrics (described in Section 3.1) to give an indication of the performance of the algorithm.

3. Simulation Testing and Results

This section first defines the test objectives and performance metrics. Then, it describes the test scenarios and discusses the results obtained.

3.1 Objectives and Performance Metrics

One of the two main objectives of the simulation tests was to assess the ability of the algorithm to create conflict-free routes which minimise flight schedule delays and taxi times. The performance of the algorithm in this regard was measured using the following metrics:

- **Average Taxi Time Delay** (TTD_{avg}): This is the average time delay accumulated by the aircraft while taxiing and is given by:

$$TTD_{avg} = \frac{\sum_{i=1}^{N_{airc}} TTD_i}{N_{airc}} \tag{12}$$

  where N_{airc} is the number of aircraft and TTD_i is the TTD for aircraft i.

- **Average Start Time Delay** (STD_{avg}): This is the average time delay accumulated by the aircraft while waiting next to the runway (for arrivals) or at the stand (for departures) and is given by:

$$STD_{avg} = \frac{\sum_{i=1}^{N_{airc}} STD_i}{N_{airc}} \tag{13}$$

  where STD_i is the STD for the aircraft i.

- **Percentage of Delayed Aircraft** (DA): This is the percentage of aircraft whose start time is delayed and is given by:

$$DA = \frac{\sum_{i=1}^{N_{airc}} TF_{delayed}^{i}}{N_{airc}} \times 100 \tag{14}$$

  where $TF_{delayed}^{i}$ is equal to 1 if $STD_i > 0$, and 0 otherwise.

The other main objective of the simulation tests was to assess the ability of the algorithm to maximise the number of aircraft towing operations and to use the available tow trucks. The performance of the algorithm in this regard was measured using the following metrics:

- **Percentage of Towing Time** (TT): This is the percentage of taxiing time of the aircraft during which they are towed and is given by:

$$TT = \frac{\sum_{i=1}^{N_{airc}} (ATT_i \times TF_{towing}^{i})}{\sum_{i=1}^{N_{airc}} (ATT_i)} \times 100 \tag{15}$$

  where TF_{towing}^{i} is equal to 1 if the aircraft i is towed, and 0 otherwise.

- **Percentage of Towed Aircraft** (TA): This is the percentage of towed aircraft and is given by:

$$TA = \frac{\sum_{i=1}^{N_{airc}} TF_{towing}^{i}}{N_{airc}} \times 100 \tag{16}$$

- **Percentage of Average Tow Trucks Utilisation Time** ($TTUT_{avg}$): This is the percentage of the simulation time during which the tow trucks are used (on average) and is given by:

$$TTUT_{avg} = \frac{\sum_{i=1}^{N_{tugs}} NPT_i}{ST \times N_{tugs}} \times 100 \tag{17}$$

  where NPT_i is the total time during which a tow truck is not parked (as indicated in its TST) and ST is the total simulation time.

3.2 Test Scenarios

The system was tested in a simulated environment using airfields at four different airport locations. For each simulation a number of critical parameters were varied, specifically the runways in use, the traffic levels and the ratio of tow trucks to aircraft movement. A total number of 1,698 simulation runs were conducted. The simulation time *ST* was set equal to 1 hour resulting in the number of aircraft in each simulation being equal to the number of aircraft movements per hour. For each simulation a randomized flight schedule was generated taking into account the selected traffic levels and the duration of the simulation. The system was tested on two scenarios.

Test Scenario 1 was designed to test the performance of the algorithm according to the metrics TTD_{avg}, STD_{avg} and *DA*. Since these metrics are related to the *Flight Dispatcher* module – which is executed before and independently of the allocation of tow trucks – the percentage of tow trucks in *Test Scenario 1* was set equal to zero. While the minimum number of aircraft movements per hour was set equal for all airports, the maximum was set in proportion to the size of each airport. For each airport, simulations were repeated for levels of traffic ranging from the minimum number of aircraft per hour to the maximum number of aircraft per hour, in increments of two aircraft per hour. The parameters of *Test Scenario 1* are shown in Table 6.

Table 6 - Airfield parameters used for the *Test Scenario 1*.

Airport name (with IATA code)	Airport size	Active runways per simulation	Aircraft per hour		Percentage of tow trucks (%)
			Minimum	Maximum	
Malta International Airport (MLA)	Small	1	20	40	0
Toulouse–Blagnac Airport (TLS)	Medium	2		60	
Ben Gurion Airport (TLV)	Medium	1		60	
Dallas/Fort Worth International Airport (DFW)	Large	4		80	

Test Scenario 2 was designed to test the performance of the algorithm according to the metrics *TT*, *TA* and $TTUT_{avg}$. For each airport, the level of traffic was fixed and simulations were repeated for different percentages of tow trucks (between 0 and 50%, in increments of 5%), where the percentage of tow trucks is relative to the number of aircraft per hour. For instance, a percentage of tow trucks of 50% means that there is a tow truck for every two aircraft. The parameters of *Test Scenario 2* are shown in Table 7.

Table 7 - Airfield parameters used for Test Scenario 2.

Airport name (with IATA code)	Airport size	Active runways per simulation	Aircraft per hour	Percentage of tow trucks (%)	
				Minimum	Maximum
Malta International Airport (MLA)	Small	1	30	0	50
Toulouse–Blagnac Airport (TLS)	Medium	2	40		
Ben Gurion Airport (TLV)	Medium	1	40		
Dallas/Fort Worth International Airport (DFW)	Large	4	50		

3.3 Results and Discussion

After obtaining all simulation results and computing the performance metrics, the results for each group of runways in the same airport were averaged for ease of reporting. More specifically, in TLS and in DFW the results are averaged between two groups of runways (consisting of 2 runways in TLS and of 4 runways in DFW), while the results in MLA (which has 4 runways in total) and in TLV (which has 6 runways in total) are averaged between each single runway. This section therefore presents the average results obtained.

Figures 4-6 show the results of Test Scenario 1. Figure 4 shows TTD_{avg} for different levels of traffic for each airport under test. In all cases, *TTD_{avg}* is relatively low and shows a weak association with the volume of traffic. This result was expected since the traffic levels being simulated are within the handling capacity of the airfields and therefore no significant changes to the ideal (shortest) taxi route were required. It is interesting to note that, despite the different dimensions of the airports, the differences of values of *TTD_{avg}* between the airports are very small (generally less than 20 seconds), except for TLV, which has slightly higher values. This might be due to the airport's geometry; for instance, DFW is large, but most of the gates are situated in the middle of the aerodrome. In contrast, MLA is small, but a number of gates are distant from some of the runways. Also, DFW has an extensive taxiway network – which provides the opportunity to find several alternative routes to the ideal one – whereas, in MLA, the number of taxiways is very limited and aircraft might be forced to follow long detours to avoid potential conflicts with other vehicles.

Figures 5 and 6 show *STD_{avg}* and *DA*, respectively, for different levels of traffic in each of the airports under test. The figures show similar trends: for each airport, *STD_{avg}* and *DA* increase with the level of traffic first moderately, then sharply. The trend of each airport also depends on the airport's size and geometry. For instance, in the case of MLA, the small dimensions of the airport cause the values to increase significantly for levels of traffic exceeding 30 aircraft per hour. While TLV has a complicated geometry (e.g. many taxiways cross the runways) and only one active runway at a time, TLS features a simple geometry and two active runways; therefore, it has lower average delays and a smaller percentage of delayed aircraft compared to TLV for similar levels of traffic.

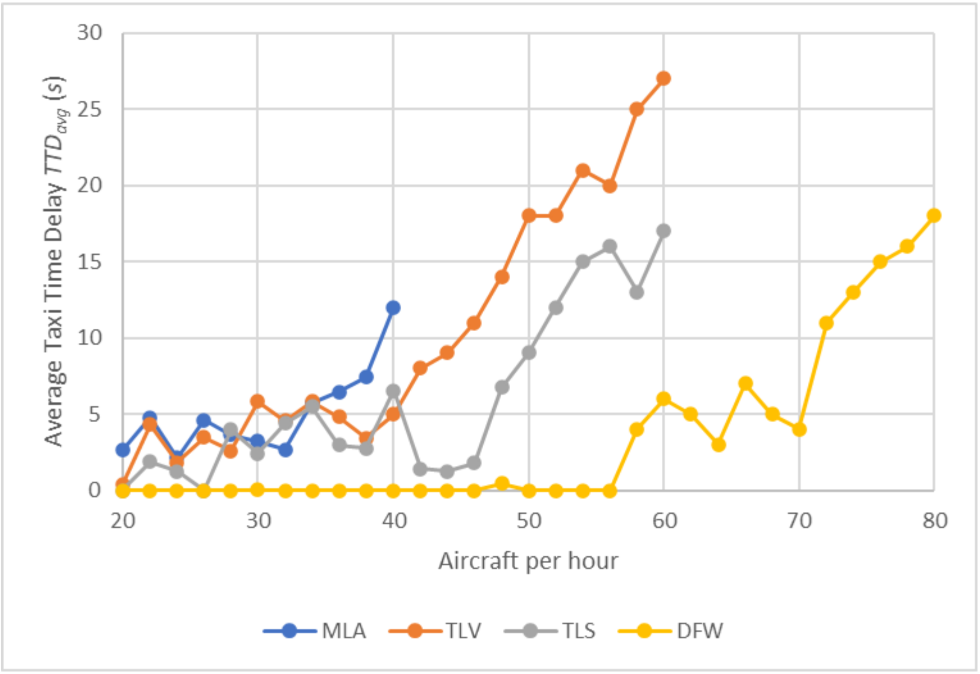

Figure 4 - *TTD_{avg}* with percentage of tow trucks equal to 0%.

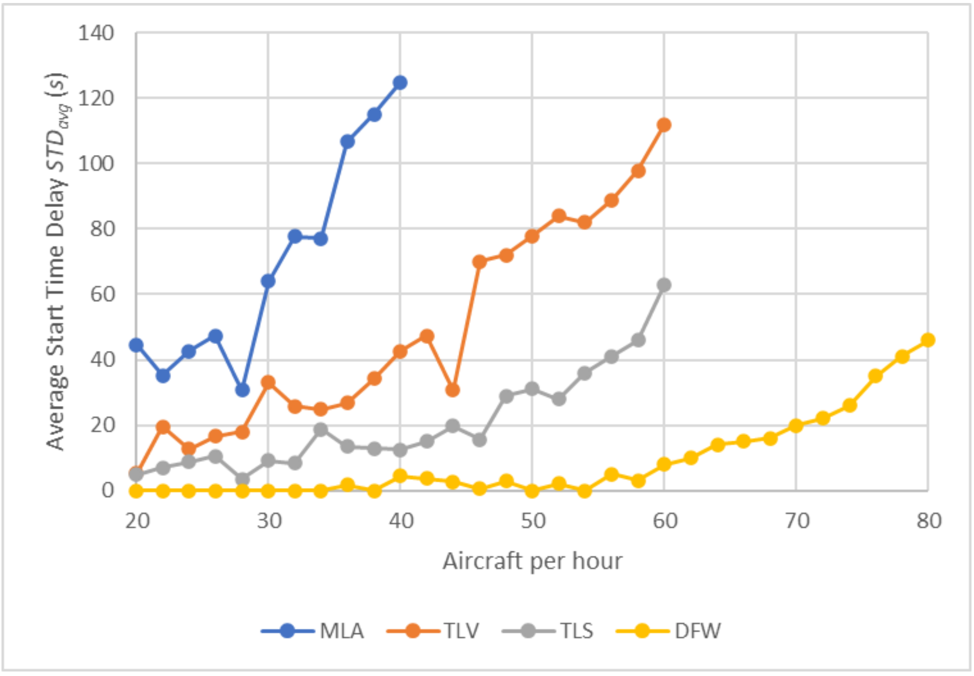

Figure 5 - *STD_{avg}* with percentage of tow trucks equal to 0%.

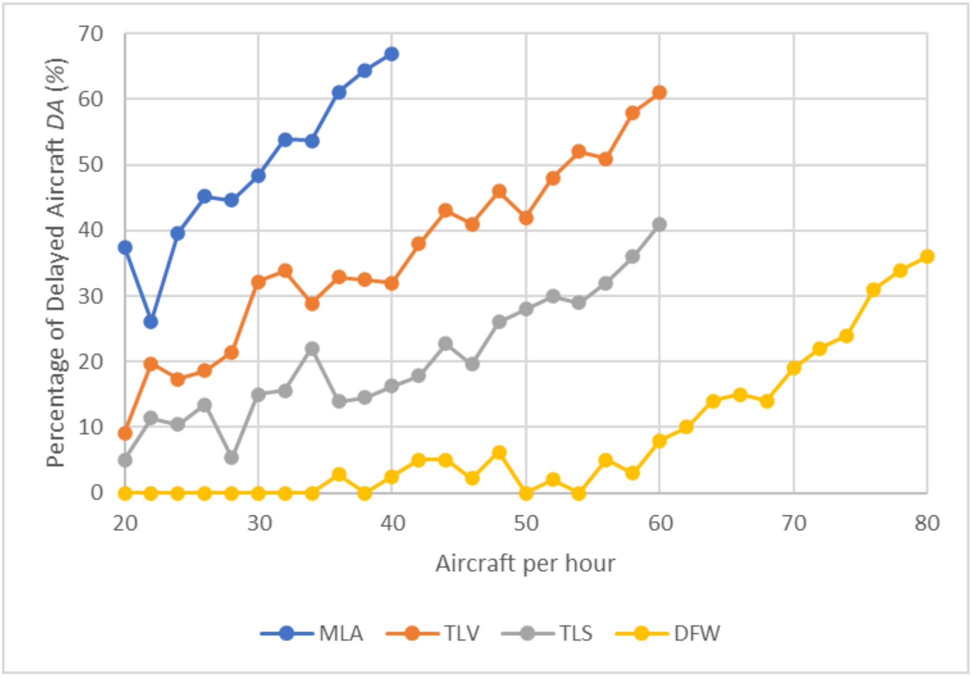

Figure 6 - *DA* with percentage of tow trucks equal to 0%.

Figures 7-9 show the results of *Test Scenario 2*. Since the results are comparing airfields with different traffic handling capacities, it was decided to report results obtained using around 75% of the traffic handling capacity for each airfield under evaluation.

Figures 7 and 8 show *TT* and *TA*, respectively, for different percentages of tow trucks for each airport. Both percentages initially increase with an increasing percentage of tow trucks but then flatten out. In both cases, when the percentage of tow trucks exceeds approximately 25%, 80% (or more) of the traffic is handled by the tow trucks. This means that only 20% (or less) of the aircraft have to taxi using their main engines and there is no significant improvement when the percentage of tow trucks is increased beyond 25%.

Figure 9 shows $TTUT_{avg}$ for different percentages of tow trucks where, as expected, $TTUT_{avg}$ steadily decreases for all airports as the percentage of tow trucks is increased. Interestingly, $TTUT_{avg}$ never exceeds 50%; one of the reasons for this could be the fact that the tow trucks have to recharge their battery. This clearly shows that battery performance is a crucial factor in tow truck-based electric taxi operations. Apart from using fast charging tow trucks, this utilisation value can be improved by adding a feature to the system which assigns tow trucks not only when these are parked in a depot, but also whilst returning to a depot after a previous mission. In this case, the remaining battery charge needs to be confirmed adequate for the mission being assigned.

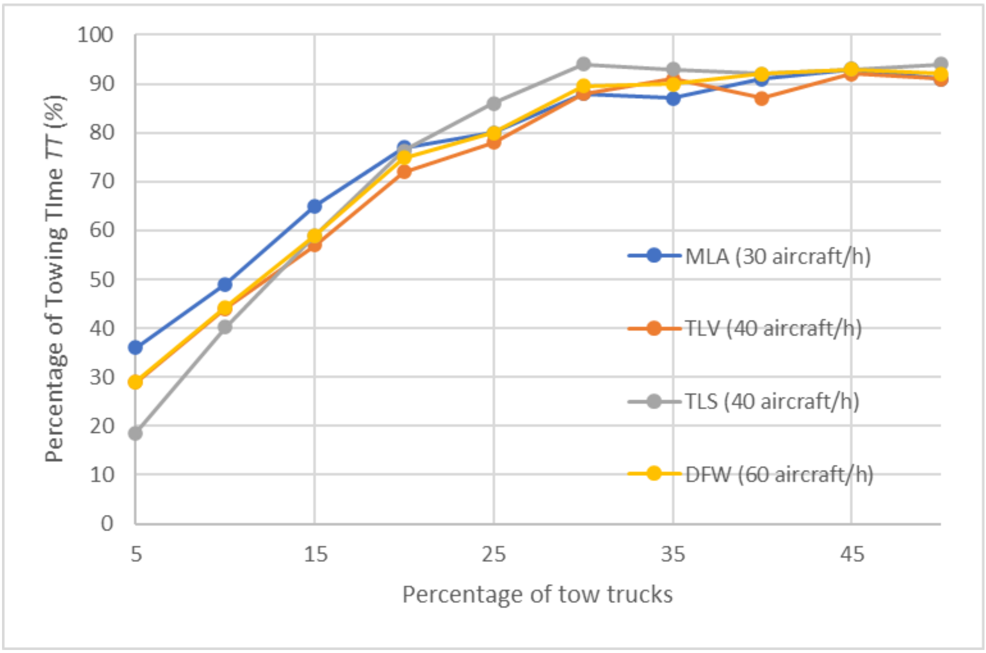

Figure 7 - Percentage of Towing Time (*TT*).

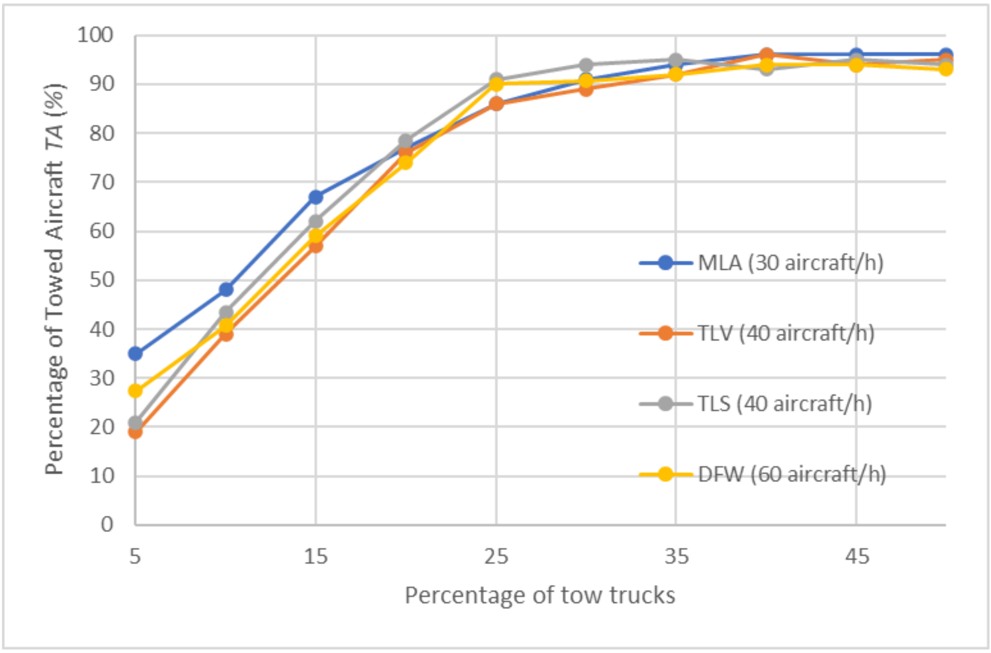

Figure 8 - Percentage of Towed Aircraft (*TA*).

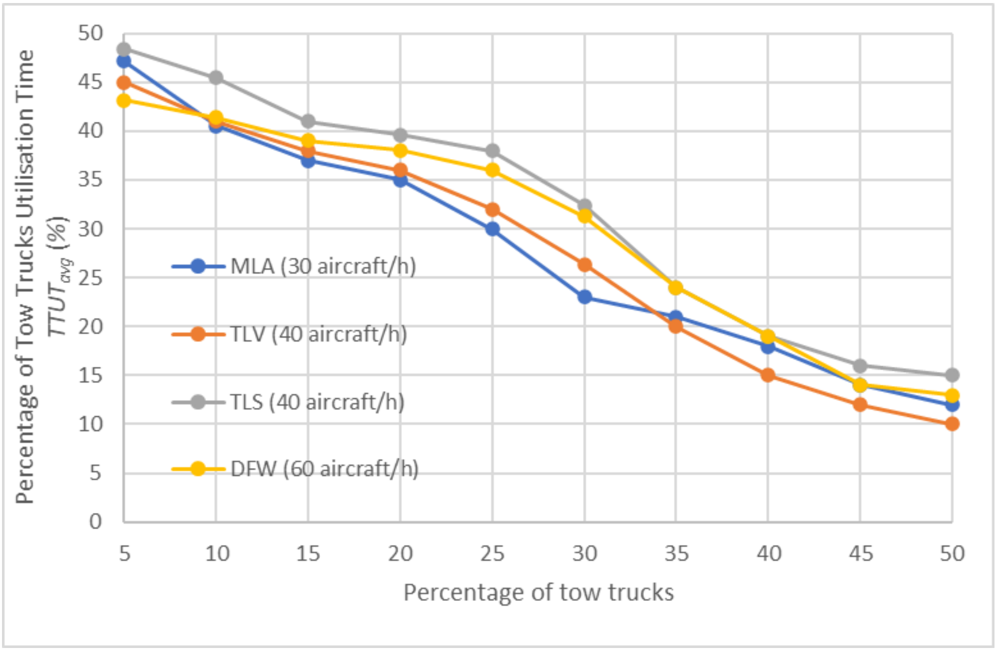

Figure 9 - Percentage of Tow Trucks Utilisation Time ($TTUT_{avg}$).

4. Conclusion and Future Work

This paper introduced an algorithm to automate and optimise taxi operations using autonomous tow trucks. The algorithm generates conflict-free routes for the aircraft and tow trucks; minimises taxi-related delays; and maximises the use of the tow trucks. The proposed algorithm was tested in a simulation environment and its performance was assessed on the basis of various metrics. The results show that the algorithm is effective in limiting delays with respect to the flight schedule (even for high volumes of traffic) and in exploiting the use of the tow trucks.

In terms of quantity, the introduced delays depend on the size and layout of the airfield, together with the levels of traffic selected for the simulation. The maximum taxi time delay introduced when compared to the shortest path was around 30 seconds and the maximum average delay in the taxi start time was around 130 seconds. The percentage of delayed aircraft varied significantly with the size of the airport, with almost 70% of the aircraft suffering some delay in the case of the smallest airport being considered (i.e. MLA).

With regards to the tow truck allocations, the results indicate that when the number of tow trucks exceeds approximately 25% of the number of aircraft per hour, most of the aircraft (around 80%) are serviced by the tow trucks, with only around 20% of the taxiing aircraft having to use their main engines. At this level of usage, the tow truck utilisation time was found to be around 30%. Increasing the size of the tow truck fleet presented no major improvements, beyond these values.

In future work, the algorithm will be modified such that it is able to assign a tow truck to a new towing mission as soon as it detaches from an aircraft, as long as it has sufficient battery charge. The algorithm will also be modified such that, in the event that no conflict-free routes are found for any available tow truck to reach an aircraft, the algorithm will be able to make slight adjustments to the *AST* of that aircraft in order to produce a conflict-free route for the tow trucks.

Another area of future work is the design of a simulator which models uncertainties related to taxi operations, including uncertainties in vehicle speed; aircraft arrival and departure times; battery charging and discharging rates; etc. This simulator will make it possible to check the sensitivity and robustness of the algorithm to various uncertainties and unforeseen changes.

5. Contact Author Email Address

To reach the contact author (Ing. Stefano Zaninotto), send an email to: stefano.zaninotto@um.edu.mt

6. Copyright Statement

The authors confirm that they, and/or their company or organization, hold copyright on all of the original material included in this paper. The authors also confirm that they have obtained permission, from the copyright holder of any third party material included in this paper, to publish it as part of their paper. The authors confirm that they give permission, or have obtained permission from the copyright holder of this paper, for the publication and distribution of this paper as part of the ICAS proceedings or as individual off-prints from the proceedings.

References

[1] European Commission. Flightpath 2050, Europe’s Vision for Aviation. [Online]. Available: https://www.acare4europe.org/sites/acare4europe.org/files/document/Flightpath2050_Final.pdf [Accessed: 8 June 2022].

[2] [Online]. Available: https://ec.europa.eu/clima/policies/transport/aviation_en [Accessed: 8 June 2022].

[3] Fleuti E and Maraini S. Taxi-Emissions at Zurich Airport. 2017.

[4] Guo R, Zhang Y and Wang Q. Comparison of emerging ground propulsion systems for electrified aircraft taxi operations. Department of Civil and Environmental Engineering, University of South Florida, Florida, USA.

[5] [Online]. Available: https://www.wheeltug.com/. [Accessed: 8 June 2022].

[6] Norris G. Honeywell/Safran Joint Venture Tests Electric Taxiing. 24 June 2013. [Online]. Available: https://aviationweek.com/awin/honeywellsafran-joint-venture-tests-electric-taxiing. [Accessed 8 June 2022].

[7] Taxibot. [Online]. Available: https://www.taxibot-international.com [Accessed 8 June 2022].

[8] Zaninotto S, Gauci J, Farrugia G and Debattista J. Design of a Human-in-the-Loop Aircraft Taxi Optimisation System Using Autonomous Tow Trucks. University of Malta, Malta, 2019.

[9] Zaninotto S, Gauci J and Zammit B. A Testbed for Performance Analysis of Algorithms for Engineless Taxiing with Autonomous Tow Trucks. University of Malta, Malta, 2021.