A Multi-shot ASP Encoding for the Aircraft Routing and Maintenance Planning Problem *

Pierre Tassel[1], Martin Gebser[1**], and Mohamed Rbaia[2]

[1] University of Klagenfurt, Klagenfurt, Austria
{pierre.tassel,martin.gebser}@aau.at
[2] Amadeus IT Group, Villeneuve-Loubet, France
mohamed.rbaia@amadeus.com

Abstract. The Aircraft Routing and Maintenance Planning problems are integral parts of the airline scheduling process. We study these relevant combinatorial optimization problems from the perspective of Answer Set Programming (ASP) modeling and solving. In particular, we contrast traditional single-shot ASP solving methods to a novel multi-shot solving approach, geared to rapidly discover near-optimal solutions to sub-problems of increasing granularity. As it turns out, our multi-shot solving techniques can heavily speed up the optimization process without deteriorating the solution quality in comparison to single-shot solving. We also provide a customizable instance generator and a solution viewer to facilitate intensive investigation of Aircraft Routing and Maintenance Planning as a benchmark problem. Our multi-shot solving techniques are however not limited to this benchmark alone, and the underlying ideas can be naturally applied to a variety of scheduling problems.

1 Introduction

Combinatorial optimization problems are usually solved in a single shot, but sometimes, we can decompose them into sub-problems (for example with a time-window approach [17]) that are then solved with some kind of local search. In this paper, we present an approach to solve the Aircraft Routing and Maintenance Planning problem in Answer Set Programming (ASP) [4,9] by decomposing it with a time-window approach using a paradigm called multi-shot solving [10]. Multi-shot ASP solving methods have already been successfully applied in areas like automated planning [7], automated theorem proving [11], human-robot interaction [6], multi-robot (co)operation [19] and stream reasoning [15]. Presumably closest to our work, proposing multi-shot solving techniques to successively increase the granularity of hard combinatorial optimization problems, is the *Asprin* system [3] that implements complex user preferences by sequences of queries, yet without decomposing the underlying problem representation.

An airline operator scheduling process is divided into six major steps [13], sometimes seen as independent sub-problems, sometimes with or without communication between the sub-problems.

* This paper also appeared at the 13th Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP 2020).

** also affiliated with the Graz University of Technology

1. **Flight Schedule Preparation**: The airline designs a set of flights to perform, choosing which airports to serve, at which time period, and the frequencies of visits to maximize the profit.
2. **Fleet Assignment**: Define the type of aircraft that should perform each specific flight, where each type of aircraft has different characteristics: total number of seats, fuel consumption, number of first-class seats, number of crew members needed to perform the flight, etc.
3. **Aircraft Routing**: Each flight gets a specific aircraft assigned to it, and a sequence of flights assigned to the same aircraft forms a route. We need to respect some constraints like airport turnaround time (called **TAT**): This is the minimum time on ground between two consecutive flights needed to perform operations preparing the aircraft for the next flight. Another condition is **airport continuity**: The start airport of an aircraft's next flight is the same as the end airport of the previous flight.
4. **Maintenance Planning**: We assign **maintenance slots** to each aircraft to respect limits defined by a certain number of cycles (i.e., flights), the number of hours from the last maintenance or hours of flight. Maintenance can only be performed at specific airports (with required equipment and skill-set), they have a minimal duration and they need to be performed before the aircraft has reached the limit. A good solution usually maximizes the usage of the aircraft.
5. **Crew Scheduling**: Assign a crew to cover each flight, while respecting all legal restrictions. A good solution tries to fulfill all crew members' preferences in addition.
6. **Disruption Recovery**: Manage the disruption events happening on the day of operation as a result of unforeseen events such as bad weather conditions, crew strikes, aircraft mechanical failures, airport closure, etc. and minimize the impact of different actions like cancellations, delays, diversions, etc. on passenger services.

We aim to solve the Aircraft Routing and Maintenance Planning together, considering one type of maintenance to be performed every seven days on each aircraft. It is possible to add other types of maintenances that deal with different due limits (e.g., cycles and hours of flight) without too much overhead, but it is out of the scope of this paper. Our encoding is able to find a solution when there is no perfect route that respects all **TAT** constraints. We show how to address this problem with ASP using multi-shot solving, and we implement our approach with *Clingo* [8].

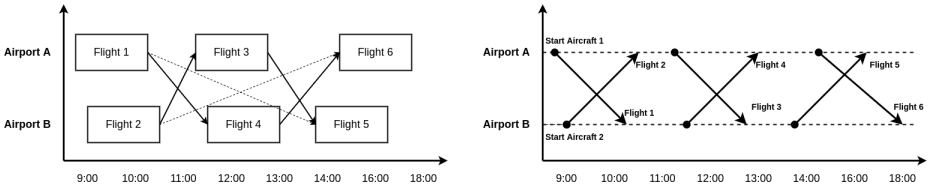
This paper is organized as follows. In Section 2, we begin with brief introductions of solving techniques for Aircraft Routing and Maintenance Planning from the literature and of ASP. Section 3 presents our customizable instance generator along with a solution viewer enabling comprehensive benchmarking. In Section 4, we develop and experimentally evaluate a variety of multi-shot ASP solving techniques for near-optimal Aircraft Routing and Maintenance Planning. Finally, Section 5 concludes the paper.

2 Background

In this section, we first introduce the works previously done in Aircraft Routing and Maintenance Planning, and then we give a brief introduction on ASP.

2.1 Aircraft Routing and Maintenance Planning

Aircraft Routing is usually considered as a feasibility problem, which is NP-hard and can be reduced to a multi-commodity flow problem [18]. Its combination with Maintenance Planning can be viewed as an Euler tour problem with side constraints [14].



(a) Flight connection network (b) Time-space network

Fig. 1: Two principal models used for Aircraft Routing and Maintenance Planning

Either kind of problem is usually solved using mixed integer programming, formulated as multi-commodity flow problem with one commodity per aircraft and side constraints related to maintenance allocation [12,16]. There are two principal models (where maintenance slots can be understood as flights from and to the same airport):

1. **Flight connection network** (Fig. 1a): In abscissa the time, in ordinate the airport, each flight is a node, and there is an arc between two flights if they are compatible, i.e., the end airport of flight A is the same as the start airport of flight B, and flight A ends before the departure of flight B [12].
2. **Time-space network** (Fig. 1b): In abscissa the time, in ordinate the airport, each node is an airport at a given time, i.e., flight start or end. Also, there is an arc between two nodes if there is a corresponding flight from one airport to another [20].

2.2 Answer Set Programming

Answer Set Programming (ASP) is a declarative paradigm oriented towards solving combinatorial problems [4,9]. We represent a problem as a logic program, and the solutions are given by models called answer sets. ASP systems like *Clingo* [8] and *DLV* [5] use a grounder to replace variables by constants and a solver to search for answer sets.

A logic program consists of atoms, literals and rules. An atom is a proposition, literals are atoms with or without default negation in front of them, and a *rule* is an implication

$$a_1,\ ...,\ a_n \leftarrow b_1,\ ...,\ b_m,\ not\ c_1,\ ...,\ not\ c_o.$$

where $a_1, ..., a_n$ is a disjunction of literals called *head*, and $b_1, ..., b_m, not\, c_1, ..., not\, c_o$ is the *body*. From the body, we can derive that the head must be true. A special case of disjunctive rules with head $a_1, not\ a_1$ are *choice* rules written as

$$\{a_1\} \leftarrow b_1,\ ...,\ b_m,\ not\ c_1,\ ...,\ not\ c_o.$$

This means that a_1 can but need not be derived from the body of the rule. A rule with an empty head is called a *constraint*, and it forbids the body to be true:

$$\leftarrow b_1,\ ...,\ b_m,\ not\ c_1,\ ...,\ not\ c_o.$$

Multi-shot ASP solving is an iterative approach geared for problems where the logic program is continuously changing [10]. In this paper, we use multi-shot solving to decompose the optimization process into a sequence of queries of increasing complexity.

3 Instance generator

The following subsections discuss how instances for our benchmarks are generated, using the generator provided at [1]. We start by introducing the parameters of the instance generator, then explain the allocation of maintenance slots in order to obtain a draft solution, further describe how a cost indicating the draft solution's quality is calculated, and finally we present a visual solution format.

3.1 Parametric generation

We have developed an instance generator that is able to create random instances along with draft solutions, configured with the following parameters:

- number of aircrafts
- number of airports
- maintenance due limit
- number of airports able to perform the maintenance
- length of maintenance
- average number of flights per aircraft
- average length of flights
- average length of flights' **TAT**
- average ground time between two flights

The flights per aircraft, flight lengths, **TAT**s and ground times are generated following a truncated normal distribution with a parametric mean, standard deviation, min and max value. We also prevent the creation of flights with the same origin and destination but different flight length or **TAT**, so that the length and **TAT** will be the same for all flights from A to B.

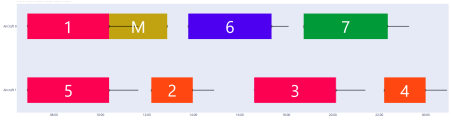
3.2 Maintenance allocation

Initial maintenance counters, expressing the time left before performing maintenance at the start of a route, are generated following a truncated normal distribution with a mean of 3.5 days, a standard deviation of 1 day, a minimum of 0 and a maximum of 6 days. While the generator builds the flight routes of a solution, it also places maintenance slots to ensure that the solution is feasible from a maintenance perspective. To do so, when an aircraft has reached at least 50% usage (i.e., 3.5 days for our 7 days maintenance), a maintenance slot is included with a probability of the usage plus a random value uniformly sampled between 0 and 0.5, or 1 if the usage is above 90%. In case the end airport of the previous flight is incompatible with the maintenance, we change the destination to a compatible airport, picked randomly among the airports able to perform the maintenance. Moreover, we add the length of the maintenance to the ground time between consecutive flights (meaning that we can have more ground time than needed).

The draft solution generated along with an instance witnesses that all flights can be routed and maintenance due limits be respected. Instead of the entire routes, the generated instance fixes the first flight for each aircraft and dates of remaining flights only, accompanied by information about initial maintenance counters, airports at which maintenance can be performed, the maintenance length and due limit. That is, allocating aircrafts to all but the first flights of routes and incorporating maintenance slots is subject to Aircraft Routing and Maintenance Planning.

3.3 Solution cost

Along with the actual instance, our generator reports its draft solution together with a cost indicating the solution quality. The latter is calculated as the sum of cost 500 for each **TAT** violation (i.e., too short turnaround time) and 101 for each maintenance slot, where the ratio reflects a higher priority of avoiding **TAT** violations and the odd cost of 101 is taken to facilitate reading off the number of maintenance slots contained in the draft solution. This information can be used for analysis, considering that the draft solution does not include **TAT** violations and is thus optimal from a flight routing perspective, yet potentially sub-optimal from a maintenance perspective. However, the



(a) Gantt chart for a small instance

```
flight(1, 1, 366701, 3, 379361).  tat(1, 4520).
flight(2, 3, 385901, 1, 392321).  tat(2, 3300).
flight(3, 1, 401861, 3, 414521).  tat(3, 4520).
flight(4, 3, 421961, 1, 428381).  tat(4, 3300).
flight(5, 1, 366417, 3, 379077).  tat(5, 4520).
flight(6, 3, 391617, 2, 404517).  tat(6, 2640).
flight(7, 2, 409497, 1, 422517).  tat(7, 3300).
first(1, 1).  first(5, 2).
maintenance(seven_day).
airport_maintenance(seven_day, 3).
length_maintenance(seven_day, 9000).
start_counter(seven_day, 366701, 416288, 1).
start_counter(seven_day, 366417, 470841, 2).
limit_counter(seven_day, 604800).
```

(b) ASP facts for the instance shown in Fig. 2a

Fig. 2: Chart and facts for an Aircraft Routing and Maintenance Planning instance

quality of the draft solution can be assumed to be rather good, given that the usage of each aircraft is at more than 50% before maintenance is performed.

3.4 Solution viewer

To inspect a solution, we support exporting a graphical representation of it as Gantt chart (Fig. 2a and Fig. 4). Every flight is represented by a bar, using a unique color for each pair of origin and destination airport, and maintenance slots after flights are indicated similarly. The tail at the right of each (non-maintenance) bar represents the **TAT** of a flight, and a next flight covering part of this tail would point out a **TAT** violation. Each row gives the route of a separate aircraft, with the first flight on the very left and further flights and maintenance slots to the right.

4 ASP-based Aircraft Routing and Maintenance Planning

In this section, we present our multi-shot ASP encoding for Aircraft Routing and Maintenance Planning. Then we specify the parameters used to furnish a benchmark suite by means of the generator described in Section 3. The remaining subsections introduce a variety of hyper-parameters for multi-shot ASP solving and experimentally evaluate their impact on the solution quality and convergence of the optimization process.

4.1 Problem encoding

As customary in ASP, we model Aircraft Routing and Maintenance Planning by facts describing a problem instance along with a general first-order encoding specifying (optimal) solutions. Our modeling approach follows the idea of flight connection networks,[3] where two flights can be connected if they are compatible (i.e., flight A arrives before flight B departs from the destination airport of flight A). In the following, we present a simplified yet logically similar version of the full encoding provided at [1].

Fig. 2a sketches (the optimal solution to) the small Aircraft Routing and Maintenance Planning instance described by the facts in Fig. 2b. We have the flights `1` to `7`, declared by facts of the `flight`/5 predicate whose first argument is the flight identifier, the second stands for the start airport, the third for the start time, the fourth for the destination airport and the fifth for the arrival time. For each of the seven flights, a fact of the `tat`/2 predicate provides the **TAT** required before the next flight on the route of some aircraft, e.g., `4520` time units (resembling about 75 minutes) for flight `1`. Two facts of the

[3] We have also devised prototype encodings based on time-space networks and observed drastically increased difficulty of finding feasible routings that incorporate all flights. Hence we chose flight connection networks as basic principle of problem encodings to elaborate further.

`first`/2 predicate indicate that flight `1` is the first on the route of aircraft `1`, and similarly flight `5` for aircraft `2`. The remaining facts address conditions for a maintenance kind labeled `seven_day`, declared by a fact of `maintenance`/1. Such maintenance can be performed at airport `3` and requires at least `9000` units of ground time (amounting to 2.5 hours), as expressed by facts of the predicates `airport_maintenance`/2 and `length_maintenance`/2. The two facts of `start_counter`/4 denote initial time periods in which the `seven_day` maintenance is (still) covered: This period stretches from time `366701` to `416288` for aircraft `1`, and from `366417` to `470841` for aircraft `2`. Finally, the fact of the `limit_counter`/2 predicate expresses that `604800` time units (7 days) get covered when `seven_day` maintenance is performed for an aircraft. The (optimal) routing, depicted in Fig. 2a, happens to be such that aircraft `1` takes the flights `1`, `6` and `7` with a maintenance slot after flight `1`, while aircraft `2` does the remaining flights in the order `5`, `2`, `3` and `4`.

Our multi-shot ASP encoding in Fig. 3 starts by defining constants for levels and weights to penalize **TAT** violations and maintenance slots along the routes of aircrafts. In addition, the constant `time_window` is crucial for when to consider compatible flight connections in a routing, and the value `3600` expresses that the gap admitted between the arrival and departure of connected flights shall be successively increased by windows of one hour. This gap is reflected by the `TIME_G` and `WINDOW` arguments in atoms of the `compatible`/6 predicate. E.g., we derive the atoms `compatible(1,3,379361,2,6540,2)` and `compatible(1,3,379361,6,12256,4)`, indicating a ground time of `6540` time units between the arrival of flight `1` at time `379361` and the departure of flight `2` from airport `3`, while this ground time amounts to `12256` time units for flight `6`. Given the window size of `3600` time units, the last argument in both atoms expresses that the potential connection between flight `1` and `2` shall be considered from the second step on during multi-shot solving, and the connection continuing with flight `6` becomes admissible from the fourth step on.

The second kind of auxiliary atoms derived from the facts of an instance, those of the `maintainable`/5 predicate, provide flights `FLIGHT1` with their arrival `TIME` such that performing `MAINTENANCE` after them covers (later) flights whose arrival and departure times lie in the interval from `TIME_M` to `TIME_N`. For our instance in Fig. 2b, we obtain `maintainable(seven_day,1,379361,388361,984161)` and `maintainable(seven_day,5,379077,388077,983877)`, signaling the possibility of `seven_day` maintenance after flight `1` and `5`, both of which arrive at airport `3` and admit connections to later flights with more than the maintenance length of `9000` time units in-between. Unlike that, performing `seven_day` maintenance after flight `3`, which also arrives at airport `3`, would be meaningless because its single available connection with flight `4` does not include sufficient ground time, i.e., `7440` time units only, so that no `maintainable`/5 atom is derived for flight `3`.

While flight connections are to be made available step-wise during multi-shot solving, an **`#external`** declaration introduces respective atoms of the `route`/4 predicate right at the beginning. This avoids need for re-instantiating conditions expressed by **`#count`** aggregates, enforcing a routing with at most one (direct) successor per flight and exactly one predecessor for flights that are not the first on the route of any aircraft, in case new compatible connections become admissible in a step. The same applies to rules for the `assign`/2 predicate, which trace connections given by atoms of `route`/4

```
% constants for levels and weights of costs, and time window for connections
#const level_tat = 2.          #const weight_tat = 1.
#const level_maintenance = 1.  #const weight_maintenance = 1.
#const time_window = 3600.
% compatible flights with number of time window
compatible(FLIGHT1, AIRPORT_E1, TIME_E1, FLIGHT2, TIME_G, WINDOW) :-
    flight(FLIGHT1, AIRPORT_S1, TIME_S1, AIRPORT_E1, TIME_E1),
    flight(FLIGHT2, AIRPORT_E1, TIME_S2, AIRPORT_E2, TIME_E2),
    not first(FLIGHT2, _),
    TIME_G = TIME_S2 - TIME_E1, 0 <= TIME_G,
    WINDOW = TIME_G / time_window + 1.
% feasible maintenance slots after flights
maintainable(MAINTENANCE, FLIGHT1, TIME, TIME_M, TIME_N) :-
    compatible(FLIGHT1, AIRPORT, TIME, FLIGHT2, TIME_G, WINDOW),
    airport_maintenance(MAINTENANCE, AIRPORT),
    length_maintenance(MAINTENANCE, LENGTH), LENGTH <= TIME_G,
    limit_counter(MAINTENANCE, LIMIT),
    TIME_M = TIME + LENGTH, TIME_N = TIME + LIMIT.

% declare incrementally generated routing as external
#external route(FLIGHT1, FLIGHT2, TIME_G, WINDOW) :
    compatible(FLIGHT1, AIRPORT, TIME, FLIGHT2, TIME_G, WINDOW).
% enforce routing sequences that include all flights
 :- flight(FLIGHT1, AIRPORT_S, TIME_S, AIRPORT_E, TIME_E),
    #count{FLIGHT2 : route(FLIGHT1, FLIGHT2, TIME_G, WINDOW)} > 1.
 :- flight(FLIGHT2, AIRPORT_S, TIME_S, AIRPORT_E, TIME_E),
    not first(FLIGHT2, _),
    #count{FLIGHT1 : route(FLIGHT1, FLIGHT2, TIME_G, WINDOW)} != 1.
% propagate assigned planes along routing
assign(FLIGHT1, PLANE) :-
    first(FLIGHT1, PLANE).
assign(FLIGHT2, PLANE) :-
    assign(FLIGHT1, PLANE), route(FLIGHT1, FLIGHT2, TIME_G, WINDOW).

% generate maintenance slots for planes
{maintain(MAINTENANCE, TIME, TIME_M, TIME_N, PLANE)} :-
    maintainable(MAINTENANCE, FLIGHT, TIME, TIME_M, TIME_N),
    assign(FLIGHT, PLANE).
% get covered flights from initial and dynamic maintenance slots
covered(MAINTENANCE, FLIGHT, PLANE) :-
    start_counter(MAINTENANCE, TIME_M, TIME_N, PLANE),
    flight(FLIGHT, AIRPORT_S, TIME_S, AIRPORT_E, TIME_E),
    TIME_M <= TIME_S, TIME_E <= TIME_N.
covered(MAINTENANCE, FLIGHT, PLANE) :-
    maintain(MAINTENANCE, TIME, TIME_M, TIME_N, PLANE),
    flight(FLIGHT, AIRPORT_S, TIME_S, AIRPORT_E, TIME_E),
    TIME_M <= TIME_S, TIME_E <= TIME_N.
% enforce coverage of all flights
 :- maintenance(MAINTENANCE), assign(FLIGHT, PLANE),
    not covered(MAINTENANCE, FLIGHT, PLANE).
% associate costs with dynamic maintenance slots
 :~ maintain(MAINTENANCE, TIME, TIME_M, TIME_N, PLANE).
    [weight_maintenance@level_maintenance, TIME, PLANE]

#program step(t).  % incremental program to generate routing
% generate new flight connections for current time window
{route(FLIGHT1, FLIGHT2, TIME_G, t)} :-
    compatible(FLIGHT1, AIRPORT, TIME, FLIGHT2, TIME_G, t).
% enforce sufficient ground time for dynamic maintenance slots
 :- compatible(FLIGHT1, AIRPORT, TIME, FLIGHT2, TIME_G, t),
    maintainable(MAINTENANCE, FLIGHT1, TIME, TIME_M, TIME_N),
    length_maintenance(MAINTENANCE, LENGTH), TIME_G < LENGTH,
    maintain(MAINTENANCE, TIME, TIME_M, TIME_N, PLANE),
    assign(FLIGHT2, PLANE).
% associate costs with TAT violations
 :~ route(FLIGHT1, FLIGHT2, TIME_G, t), tat(FLIGHT1, TAT), TIME_G < TAT.
    [weight_tat@level_tat, FLIGHT1]
```

Fig. 3: Multi-shot ASP encoding for Aircraft Routing and Maintenance Planning

and associate each flight with its corresponding aircraft. Maintenance slots can then be scheduled for aircrafts assigned to flights indicated by the `maintainable`/5 predicate, and the arguments `TIME_M` and `TIME_N` in `maintain`/5 atoms provide the respective time period covered. The flights included in the initial interval or by performing maintenance for an aircraft are signaled by atoms of the predicate `covered`/3, where a subsequent constraint makes sure that each assigned flight is indeed covered. Reconsidering the instance in Fig. 2b, the initial `seven_day` maintenance period for aircraft `2` includes all flights that can belong to its route, while the flights `4` and `7` exceed the initial interval for aircraft `1`. Hence, aircraft `1` needs to be maintained after its first flight, as indicated by the atom `maintain(seven_day,379361,388361,984161,1)` in an (optimal) answer set. The allocation of maintenance slots is however penalized by a weak constraint, and the particular instance `:~ maintain(seven_day,379361,388361, 984161,1). [1@1,379361,1]` associates the weight `1` at level `1` with the maintenance of aircraft `1` at time `379361`.

In contrast to the upper part of the encoding in Fig. 3, the one below the **`#program`** directive is instantiated in steps during multi-shot solving, where `t` is replaced by successive integers starting from `1`. The choice rule for `route`/4 atoms, which were declared external before, then allows for taking the connections newly admitted at the current step or integer for `t`, respectively. E.g., `route(1,2,6540,2)` is introduced as a potential connection in the second step, and `route(1,6,12256,4)` in the fourth step. The subsequent constraint enforces sufficient ground time when a maintenance slot is allocated in-between two connected flights.[4] This rules out `route(1,2,6540,2)` for an aircraft subject to `seven_day` maintenance after flight `1`, as it is the case for aircraft `1` whose first flight is `1`. Hence, the routing given by an (optimal) answer set is such that the flights `1`, `6` and `7` are assigned to aircraft `1`, and aircraft `2` takes `5`, `2`, `3` and `4`. One can check that this schedule does not involve **TAT** violations, which would otherwise be penalized by a weak constraint according to the corresponding level and weight. As the greatest step associated with some flight connection in the routing happens to be `4` (indicated by the last argument in `route(1,6,12256,4)`), the multi-shot encoding requires four steps to lead to an answer set, which then describes an optimal solution for the instance in Fig. 2b.

Finally, let us note that a traditional single-shot version can be easily derived from the more sophisticated multi-shot encoding in Fig. 3 by simply omitting the `WINDOW` and `t` arguments from atoms related to flight connections, as well as dropping the **`#external`** and **`#program`** declarations. A respective single-shot encoding is also provided at [1].

4.2 Problem instances

Our benchmark suite comprises 20 random instances, generated with the parameters:

- number of aircrafts: 25
- number of airports: 30
- average number of flights per aircraft: $20 \leq X \sim \mathcal{N}(50,\ 10) \leq 80$ flights[5]
- average length of flights: $80 \leq X \sim \mathcal{N}(140,\ 120) \leq 600$ minutes
- average length of flights' **TAT**: $30 \leq X \sim \mathcal{N}(45,\ 10) \leq 60$ minutes
- average ground time between two flights: $0 \leq X \sim \mathcal{N}(240,\ 120) \leq 1000$ minutes

[4] Our full encoding at [1] includes a more general version of this constraint that is also able to deal with multiple maintenance kinds.

[5] $\mathcal{N}(\mu,\ \sigma^2)$ denotes a normal probability distribution of mean μ and standard deviation σ.

Fig. 4: Gantt chart of the draft solution used to generate an instance

Such instances are quite large in order to make optimal Aircraft Routing and Maintenance Planning challenging. While detailed inspection of a draft solution like the one displayed in Fig. 4 would be intricate, we can still observe that the numbers of flights and resulting time spans of aicrafts' routes vary significantly. As a consequence, we obtain a planning period stretching almost over one month, which necessitates the allocation of a high number of maintenance slots.

4.3 Basic multi-shot solving approach

The main bottleneck of flight connection networks is the large number of arcs when flights with long ground times in-between are taken as connection candidates, e.g., linking the first flight arriving at an airport to the last flight in the planning period departing from it. Such connections could be dropped by imposing a hard constraint on the maximum admissible ground time, yet to the risk of ruling out (optimal) solutions up to making Aircraft Routing and Maintenance Planning infeasible for tricky instances where some connection with long ground time has to be taken.

Rather than constraining ground times, our multi-shot ASP solving approach works by successively increasing the maximum ground time of the considered connections over iterations. For guaranteeing the progress to connections with longer ground times (and eventually all connections), we limit the runtime allotted for optimizing the routing and maintenance allocation in each iteration by means of the following intra-iteration stop criterion: An iteration is aborted when the empirically determined timeout of 60 seconds for finding some better solution is reached, in which case we continue to the next iteration with an increased maximum ground time of connections. The rationale of this strategy is to avoid getting stuck on infeasible sub-problems, when the admissible ground time is yet too small in the first iterations, or on (near-)optimal solutions that can neither be improved nor verified as optimal in reasonable runtime. Note that the timeout is reset to 60 seconds whenever the optimization comes up with a better solution, as we do not want to abort iterations in phases where the optimization makes progress. Upon proceeding to the next iteration, either due to timeout or search space exhaustion, we check that new connections become admissible, or increase the maximum ground time further without relaunching the optimization otherwise. Moreover, the cost of the best solution found so far, if any, is passed on as upper bound to admit better solutions only.

Our experiments consider the 20 random instances whose generation has been described in the previous subsection. As time window for increasing the maximum ground time of connections, we use the value `3600` (one hour), corresponding to the default of our encoding in Fig. 3. Unless noted otherwise, we also stick to the level `2` for weight

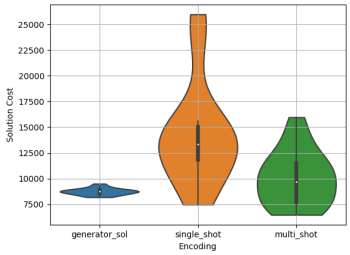
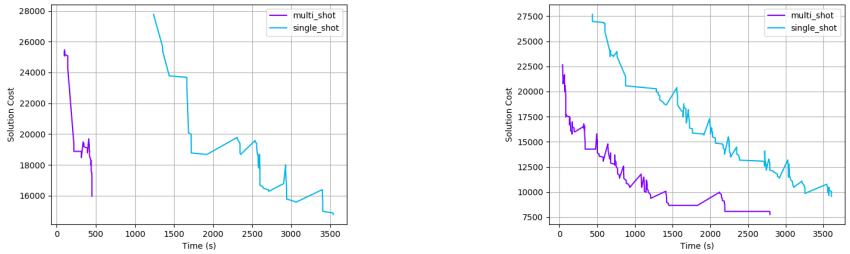


Fig. 5: Solution costs for single-shot and multi-shot solving



(a) Solution costs per runtime for instance 14 (b) Solution costs per runtime for instance 15

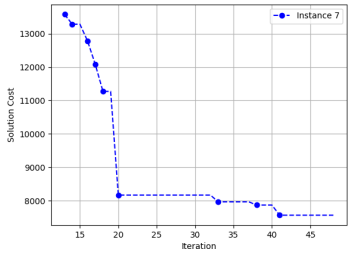
Fig. 6: Instance-wise solution costs per runtime for single-shot and multi-shot solving

constraints penalizing **TAT** violations, and the smaller level `1` for maintenance slots implies strictly lower priority of minimizing their number, where each maintenance slot or **TAT** violation is counted with the weight `1`. Note that this scheme is different from the weighted sum taken to calculate the cost of a draft solution in Section 3.3, and we reuse the latter for comparability when plotting solution costs in the sequel. All experiments were run with *Clingo* version 5.4.0, each run limited to one hour wall clock time, on an Ubuntu 18.04 machine with two 8-Core Intel Xeon E5520 processors and 48GB RAM.

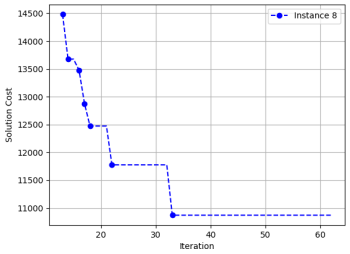
Fig. 5 plots the costs of best solutions found in one hour with traditional single-shot solving, where the full problem with all flight connections is considered, and with our (basic) multi-shot solving approach in relation to the costs of draft solutions generated together with the instances. Although multi-shot solving with its intra-iteration stop criterion merely probes the search space of sub-problems without guaranteeing that a globally optimal solution will be obtained, it usually finds better solutions than single-shot solving in the time limit, and sometimes its best solution also improves on the draft solution that is of good quality by construction.

Fig. 6a and 6b show the optimization progress in detail for two representative instances, where single-shot solving leads to a better solution for one instance and multi-shot solving for the other.[6] We observe that multi-shot solving finds its solutions much faster and gets then stuck on unsuccessful iterations aborted after 60 seconds each. Tackling the full problem by single-shot solving makes finding the first feasible routing and then achieving improvements much harder and time-consuming, so that granting

[6] Occasionally rising solution cost over time is due to the weighted sum function used for plotting the solution quality, while the optimization strictly reduces **TAT** violations in such cases and also leads to lower weighted sum values in the long run.

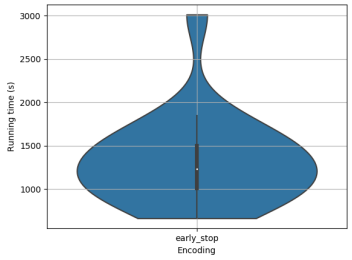


(a) Solution cost per iteration for instance 7

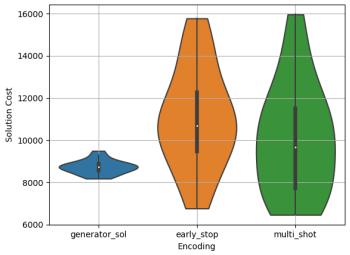


(b) Solution cost per iteration for instance 8

Fig. 7: Instance-wise solution cost per iteration for multi-shot solving



(a) Runtimes for the early-stop criterion



(b) Solution costs for the early-stop criterion

Fig. 8: Runtimes and solution costs for the early-stop criterion

non-negligible runtime is a necessity to obtain solutions of good quality. Notably, all runs exhaust the time limit of one hour due to the size and combinatorics of instances.

4.4 Early-stop multi-shot solving approach

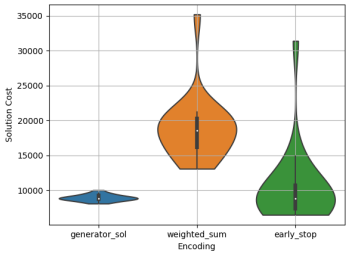
Picking again two representative instances, Fig. 7a and 7b indicate the optimization progress over the iterations of multi-shot solving, where we observe substantial improvements by step-wise increases of the maximum ground time at the beginning, followed by little and then no improvement at all for a substantial number of unsuccessful iterations aborted after 60 seconds. This suggests the addition of an inter-iteration stop criterion to avoid spending time on unpromising iterations, and our early-stop multi-shot solving approach thus aborts the entire run after timing out without any improvement for three iterations in a row. The deliberate stop of runs constitutes a trade-off between solution quality and computational efforts, and the number of three consecutive timeouts of iterations without improvement is again problem-specific and determined empirically.

The plot in Fig. 8a shows that runtimes are indeed substantially reduced by early-stop multi-shot solving, with the median around 20 minutes instead of fully exhausting the one hour per instance. Comparing the solution costs in Fig. 8b yields a rather modest quality decline in exchange for runtime savings, which can presumably be tolerated in application scenarios where the time taken for decision making is critical.

4.5 Weighted sum vs level cost function

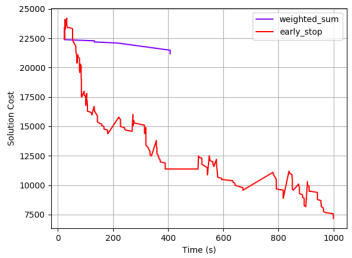
The results reported so far rely on distinct priority levels for **TAT** violations and maintenance allocation, and now we compare the performance of optimization relative to the weighted sum function given in Section 3.3. Switching to the latter can be easily done by

(a) Runtimes for weighted sum and level function

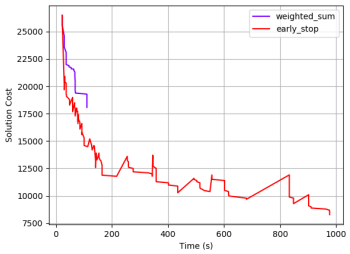


(b) Solution costs for weighted sum and level function

Fig. 9: Runtimes and solution costs for weighted sum and level function



(a) Solution costs per runtime for instance 4



(b) Solution costs per runtime for instance 12

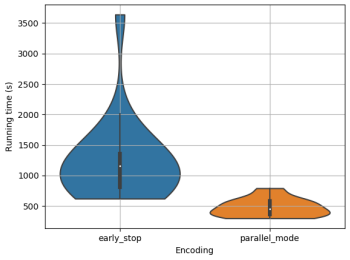
Fig. 10: Instance-wise solution costs per runtime for weighted sum and level function

setting the values for the constants `level_tat`, `level_maintenance`, `weight_tat` and `weight_maintenance` used by the encoding in Fig. 3 to `1`, `1`, `500` and `101`.

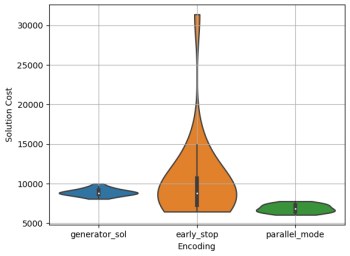
Fig. 9a and 9b plot runtimes and solution costs for early-stop multi-shot solving with either the weighted sum function or distinct priority levels to penalize **TAT** violations and maintenance slots. Switching to the weighted sum greatly reduces runtimes, yet because optimization turns out to be much harder and the three iterations in a row without improvement are reached way more quickly. Accordingly, the solution quality suffers heavily, even despite the previously considered optimization based on distinct priority levels merely approximates the weighted sum function used for plotting and now in the optimization process as well. The quick outage of improvements after more or less substantial progress in the first iterations becomes also apparent on the detailed inspections of two instances in Fig. 10a and 10b. We conjecture that higher weighted sum values due to incorporating costs from several sources at the same level complicate recognizing and discarding partial assignments that can eventually not lead to any improvement, so that more search efforts are spent on such fruitless assignments.

4.6 Parallel solving

While we merely considered single-threaded *Clingo* before, it also allows for running multiple solver threads with complementary search strategies in parallel. The remarkably reduced runtimes and solution costs obtained with eight parallel solver threads are summarized in Fig. 11a and 11b. Notably, the best solutions found by early-stop multi-shot solving with parallel threads consistently improve on the draft solutions for instances, thus showing that high-quality results can be achieved with reasonable com-

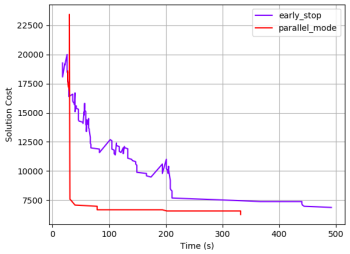


(a) Runtimes for sequential and parallel solving

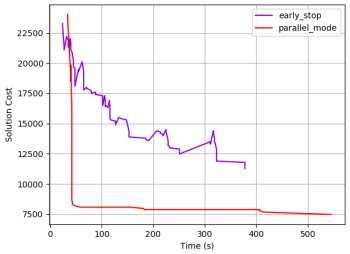


(b) Solution costs for sequential and parallel solving

Fig. 11: Runtimes and solution costs for sequential and parallel solving



(a) Solution costs per runtime for instance 0



(b) Solution costs per runtime for instance 11

Fig. 12: Instance-wise solution costs per runtime for sequential and parallel solving

putational efforts. Fig. 12a and 12b additionally plot the much more rapid optimization progress for two representative instances. This robustness is certainly related to the parallel use of complementary search strategies, also considering that the discovery of a better solution by one thread resets the timeout to another 60 seconds for all threads.

5 Conclusion

The Aircraft Routing and Maintenance Planning problem lends itself to multi-shot ASP solving based on successively increasing ground times of flight connections, given that long ground times are undesirable in practice and should thus be avoided if possible. A direct use of the incremental mode shipped with *Clingo* [10] would be (too) risky though, as it minimizes the number of iterations and can easily get stuck on hard subproblems. We instead aim at discovering near-optimal solutions in affordable time, so that approximating solution costs by means of (easier to optimize) priority levels, also found to be advantageous for shift design [2], and interrupting exhaustive iterations, as likewise done in automated planning [7], can be tolerated. The hyper-parameters we used for aborting iterations or entire runs are clearly problem-specific and need retuning when switching to another application, where related scheduling problems may benefit from similar techniques as well, so that a general tool supplying them can be valuable.

Acknowledgments. This work was partially funded by KWF project 28472, cms electronics GmbH, FunderMax GmbH, Hirsch Armbänder GmbH, incubed IT GmbH, Infineon Technologies Austria AG, Isovolta AG, Kostwein Holding GmbH, and Privatstiftung Kärntner Sparkasse. We thank the anonymous reviewers for helpful comments.

References

1. https://github.com/prosysscience/Aircraft_Scheduling
2. Abseher, M., Gebser, M., Musliu, N., Schaub, T., & Woltran, S. (2016). Shift Design with Answer Set Programming. Fundamenta Informaticae, 147(1), 1-25.
3. Brewka, G., Delgrande, J., Romero, J., & Schaub, T. (2015). asprin: Customizing Answer Set Preferences without a Headache. Proceedings of the AAAI Conference on Artificial Intelligence, 1467-1474. AAAI Press.
4. Brewka, G., Eiter, T., & Truszczyński, M. (2011). Answer Set Programming at a Glance. Communications of the ACM, 54(12), 92-103.
5. Calimeri, F., Dodaro, C., Fuscà, D., Perri, S., & Zangari, J. (2020). Efficiently Coupling the I-DLV Grounder with ASP Solvers. Theory and Practice of Logic Programming, 20(2), 205-224.
6. Chen, K., Lu, D., Chen, Y., Tang, K., Wang, N., & Chen, X. (2014). The Intelligent Techniques in Robot KeJia – The Champion of RoboCup@Home 2014. Proceedings of the Annual RoboCup International Symposium, 130–141. Springer.
7. Dimopoulos, Y., Gebser, M., Lühne, P., Romero, J., & Schaub, T. (2019). plasp 3: Towards Effective ASP Planning. Theory and Practice of Logic Programming, 19(3), 477-504.
8. Gebser, M., Kaminski, R., Kaufmann, B., Lindauer, M., Ostrowski, M., Romero, J., Schaub, T., & Thiele, S. (2019). Potassco User Guide. https://potassco.org, University of Potsdam.
9. Gebser, M., Kaminski, R., Kaufmann, B., & Schaub, T. (2012). Answer Set Solving in Practice. Morgan and Claypool Publishers.
10. Gebser, M., Kaminski, R., Kaufmann, B., & Schaub, T. (2019). Multi-shot ASP Solving with clingo. Theory and Practice of Logic Programming, 19(1), 27-82.
11. Gebser, M., Sabuncu, O., & Schaub, T. (2011). An Incremental Answer Set Programming Based System for Finite Model Computation. AI Communications, 24(2), 195-212.
12. Grönkvist, M. The Tail Assignment Problem. (2005). Ph.D. thesis, Chalmers University of Technology.
13. Jamili, A. (2017). A Robust Mathematical Model and Heuristic Algorithms for Integrated Aircraft Routing and Scheduling, with Consideration of Fleet Assignment Problem. Journal of Air Transport Management, 58, 21-30.
14. Liang, Z., & Chaovalitwongse, W. (2009). The Aircraft Maintenance Routing Problem. Optimization and Logistics Challenges in the Enterprise, 327-348. Springer.
15. Obermeier, P., Romero, J., & Schaub, T. (2019). Multi-Shot Stream Reasoning in Answer Set Programming: A Preliminary Report. Open Journal of Databases, 6(1), 33-38.
16. Orhan, İ., Kapanoğlu, M., & Karakoç, T. (2011). Concurrent Aircraft Routing and Maintenance Scheduling. Journal of Aeronautics and Space Technologies, 5(1), 73-79.
17. Ovacik, I., & Uzsoy, R. (2012). Decomposition Methods for Complex Factory Scheduling Problems. Springer.
18. Roy, K., & Tomlin, C. (2007). Solving the Aircraft Routing Problem Using Network Flow Algorithms. Proceedings of the American Control Conference, 3330-3335. IEEE.
19. Schäpers, B., Niemueller, T., Lakemeyer, G., Gebser, M., & Schaub, T. (2018). ASP-Based Time-Bounded Planning for Logistics Robots. Proceedings of the International Conference on Automated Planning and Scheduling, 509-517. AAAI Press.
20. Vaaben, B., & Larsen, J. (2015). Mitigation of Airspace Congestion Impact on Airline Networks. Journal of Air Transport Management, 47, 54-65.