Small Project July 27th, 2022

Multi-shot ASP solving for the Aircraft Maintenance Routing Problem: an Alternative Approach

Alex Della Schiava 12040036

1 Introduction

The airline scheduling process plays a key role in an airline's success. It collects a sequence of stages, each representing fundamental problems of logistical and organizational nature. Although there may be more than a way to distinguish these different stages, this report shall adopt the following structure to fully describe an airline scheduling process: i) Flight schedule preparation; ii) Fleet assignment; iii) Aircraft routing; iv) Maintenance planning; v) Crew scheduling; vi) Disruption recovery. An airline may deal with its scheduling process by taking on each stage individually or by facing combinations of different such stages. For a brief description of all the sub-problems the reader is referred to [1, Section 1] and/or [2, Chapter 2]. As for the scope of this project, the focus shall stay on *aircraft routing* and *maintenance planning*. To give a brief, preliminary description of these problems, aircraft routing is concerned with the assignment of any flight activity given by the schedule to an aircraft. On the other hand, maintenance planning is related to aircraft and their maintenance requirements. More specifically, maintenance activities are to be planned so that an aircraft satisfies its maintenance requirements during the entirety of the schedule. In this document, the combination of the two problems shall be addressed as the *Aircraft Maintenance Routing Problem* (AMRP).

Given the variety in terms of both settings and techniques with which these problems can be tackled, the related literature is rather heterogeneous. In [3], AMRP was proven to be NP-hard. In the same work, maintenance planning was proven to be solvable in polynomial-time, that is, under specific assumptions over the maintenance requirements. On the other hand, the aircraft routing problem was proven to be NP-hard [4] even when taken on individually. Aircraft routing was also proven to be equivalent to the asymmetric traveling salesman problem (ATSP) and to the search for an Eulerian tour [5]. The two equivalences hold depending on whether the schedule is represented, respectively, by means of a time-space network (ATSP) or a connection network (Eulerian tour). For the sake of brevity, this report does not discuss the two network representations, the reader is thus referred to the intuitive explanation given in [1, Section 2.1]. Both these equivalences can be extended to AMRP with the addition of side-constraints in order to take into account maintenance requirements.

Having said that, the main literature work around which this project revolves is [1], in which Tassel et al. proposed a multi-shot Answer Set Programming (ASP) [6, 7] encoding for AMRP. Differently from the previously mentioned works, AMRP is here set as an optimization problem rather than a feasibility one. The solution exploits the multi-shot paradigm to compute different solutions of increased granularity over time. This project builds upon such work, with the goal of finding alternative ways to increase granularity. Nonetheless, a side goal of this work consists in improving the overall performance of the encoding. In pursue of such goal, the work done by Grönkvist in [2] was of great influence. Grönkvist proposed a mathematical model, as well as a Constraint Programming (CP) solution for AMRP. Optimization techniques used in CP were found to fit nicely in the ASP solution developed for this project.

This report is organized as follows: Section 2 provides the required background. That is, a thorough description of AMRP and the most relevant topics coming from [1] and [2]. Section 3 illustrates the main contributions of this work: the multi-shot chunk approach for AMRP. The experimental results concerning the contributions are shown and commented in Section 4. For a closer look to the implementations and the experimental results the reader is referred to [8].

2 Background

This section provides the required background for an understanding of the results and concepts later introduced. The report assumes a basic knowledge of Answer Set Programming and its multi-shot solving paradigm.

2.1 The Aircraft Maintenance Routing Problem (AMRP)

Hereby a formal description of AMRP is given. For the sake of clarity, each definition is given together with its encoding in the ASP language. An AMRP instance can be fully described by means of a triple $\langle \mathcal{F}, \mathcal{T}, \mathcal{M} \rangle$.

$\mathcal{F}$ describes the set of all flights belonging to the schedule. Each flight is described by an ID f, the departure airport a, the departure time s, the arrival airport b, the arrival time t. Note that f uniquely identifies the flight. A flight is thus encoded by means of the following fact `flight(f,a,s,b,t).` Another key element defining a flight is its turnaround time (TAT). The TAT of a flight $f \in \mathcal{F}$ defines the time an aircraft should stay on ground after serving f before serving another flight f'. One encodes such information as `tat(f,d).` where `d` is a time interval expressed in seconds.

$\mathcal{T}$ describes the set of all available aircraft. All planning related to an aircraft's characteristics is taken care of during the earlier *fleet assignment* stage, therefore in AMRP an aircraft is solely identified by a unique ID. The encoding of an aircraft is, therefore, given by the fact `aircraft(t).`

Finally, $\mathcal{M}$ describes the set of the maintenance requirements imposed on the aircraft. A maintenance requirement is defined by its name, its expiration time and the set of airports where the related maintenance activity may be performed. These specifics are encoded respectively by the atoms:

```
maintenance(m).  length_maintenance(m,d).   airport_maintenance(m,a).
```

where `d` is a time interval expressed in seconds and `a` an airport. Following the setting adopted in [1], instances are limited to the single maintenance requirement `seven_days`, where the expiration time is, in fact, 7 days (or 604800 seconds).

Having formally described the triple $\langle \mathcal{F}, \mathcal{T}, \mathcal{M} \rangle$ it is now possible to observe how the elements of these three sets may relate to each other. Important, though previously skipped, is the set $\mathcal{F}_c \subseteq \mathcal{F}$. That is, the set of *carry-in* flight activities. These flights are key in the initial definition of the aircrafts as they denote the first flight *"carrying them into"* the schedule. More formally, each aircraft $t \in \mathcal{T}$ is mapped onto a flight $f \in \mathcal{F}_c$. These are the only flights assigned a priori to an aircraft. This relation is encoded by the fact `first(f,t).`

The current setting also requires to describe maintenance coverage for each aircraft at start time. That is, the time elapsed since the last maintenance activity at schedule's beginning. The information is encoded by fact `start_maintenance_counter(m,t,c).` where `m` is the name of the maintenance requirement, `t` an aircraft and `c` a time amount expressed in seconds.

Finally, a definition of an AMRP solution shall be given. A solution to AMRP is a surjective mapping $\sigma : \mathcal{F} \rightarrow \mathcal{T}$, that is, each flight is mapped onto (operated by) an aircraft. Note that the setting here considered assumes *any instance to have at least one such mapping*. Given an aircraft $t \in \mathcal{T}$, one is able to extract from σ all flights $f \in \mathcal{F}$ such that $\sigma(f) = t$. By chronologically ordering these flights one obtains the *route* of t, which shall be denoted as π_t. Any route $\pi_t = f_0, f_1, \ldots, f_m$ shall respect the following conditions:

- $f_0 \in \mathcal{F}_c$: any aircraft shall start its route with its carry-in flight.
- For any $i < m$, the flights f_i, f_{i+1} are *connected*. Supposing to have two flights f, f' respectively encoded as `flight(f,a,s,b,t).` and `flight(f',a',s',b',t').`, f and f' are said to be *connected* if `b = a'` and `t <= s'`. Such connection is encoded by means of the atom `compatible(f,b,t,f',g).`, where `g=s'-t` denotes the ground-time between f and f'.
- Through the entirety of route π_t, aircraft t shall satisfy the maintenance requirements $\mathcal{M}$.

Given this definition, one may rightly argue that the second condition is not strong enough to consider TATs. The reason behind this is that TAT conditions are encoded as weak constraints. They can indeed be violated albeit with a cost affecting the optimality of the solution. As is the case for TAT violations, maintenance performances do also have a cost, since performing maintenance at any available time should be deemed suboptimal.

2.2 Multi-shot ASP for AMRP

In recent years, multi-shot ASP solving has risen up as a solid and effective paradigm in the context of Logic Programming. The main perk of multi-shot solving concerns the control that it allows over a logic program. This advantage comes in the shape of two specific directives: `#program` and `#external`. The directive `#program` can be used to structure a program in different subprograms. The subprograms can later be handled in a completely modular way, both in terms of grounding and solving processes. To perform such processes successfully, conditions related to the operation of *logic programs composition* ($\sqcup$) need to be satisfied. Treating such conditions is out of this report's scope. On the other hand, directive `#external` allows for the arbitrary assignment of the truth values of specific atoms (so-called *input atoms*).

The success of multi-shot ASP lies in the ease with which these directives can be exploited to perform *problem decomposition*. One intuitive example is given by [9], where multi-shot ASP solving is applied onto the Job-Shop Scheduling problem. Here, the problem is decomposed into time-windows (logic subprograms), which are then solved individually. Finally, the overall solution is obtained by joining together the partial solutions. The complexity of the problem is thus reduced. Although optimal solutions might be excluded by the time-schedule's subdivision, techniques have been adopted to mititgate any drawback in terms of solutions' quality.

In the case of AMRP, multi-shot ASP has been exploited in a slightly different manner [1]. The logic program is again structured into different subprograms, which, instead of working on different subproblems, all work on solving the entire instance with increasing *"accuracy degrees"*. The accuracy degree refers to the amount of connections considered per flight (recall the definition of connection given in Section 2.1). This involves a partition of the set of connections of the flights. In [1], connections are partitioned in time-windows, that is, depending on the amount of hours of ground-time between first and second flight. Supposing the longest connection ground-time to be lasting k hours, the resulting program's structure is composed by k subprograms, denoted as `step(i)` for $0 < i \leq k$, where `step(i)` only considers connections with ground-time long at most i hours. Consider the following example with three flights:

```
flight(f1,"MPX",1000,"OPO",2000).   flight(f2,"OPO",3000,"MAD",4000).
flight(f3,"OPO",6000,"CDG",9000).
```

One quickly notices that flight f_1 is connected with both flights f_2 and f_3, the connections are encoded by atoms:

```
compatible(f1,"OPO",2000,f2,1000).  compatible(f1,"OPO",2000,f3,4000).
```

The longest connection takes 2 hours, giving rise to subprograms `step(1)`, `step(2)`. During its solving process, `step(1)` will ignore the second connection (since `4000>3600`), thus immediately routing f_1 with f_2. On the other hand `step(2)` shall consider both connections, and, through its solving process, find the best route. This example means to show how considering smaller amounts of connections allows for better time-performances albeit (again) with a slight drawback in accuracy and quality of the solution.

The search for the problem's solution proceeds incrementally with respect to the accuracy degree. As for the details regarding the criteria used for the incremental steps, nothing here proposed changes what shown in [1, Section 4.2]. To offer a brief insight, each incremental step gets 60 seconds to find an improved solution, in which case the timeout is reset. In the case where three consecutive steps fail to find any improvement, the solving process stops and the best solution found is returned.

2.3 Simple Aircraft Balancing Preprocessing (SABP)

Simple Aircraft Balancing Preprocessing (SABP) [2, 10] is a simple yet effective optimization technique that allows to ignore a large amount of flight connections. The technique utilizes a counter on ground aircraft at all airports belonging to the instance in different time instants. The instants in which the counter is set to zero (*zero-instants*) are of high relevance: given an airport A and a zero-instant z, no flight landing in A before z may be connected to a flight departing from A afterwards.

This is better explained through the visual example shown in Figure 1, which illustrates the timeline of arriving (entering arrows) and departing (leaving arrows) flights for airport A. Initially, the arrivals f_1, f_2 set the aircraft counter to 2. Departures f_3, f_4 give rise to the zero-instant z. Later on, arrival f_5 and departure f_6 join the timeline. Following the definition of connection, flights f_1, f_2 would be considered to be connected to flight f_6. Though one immediately notices that actually routing one of these two arrivals with f_6 implies that one flight among f_3 and f_4 shall have no available connection. In the setting here considered, this is a contradiction as any instance is assumed to have a solution. As a consequence, any connection between flights previous to z and flights following z may be dropped.

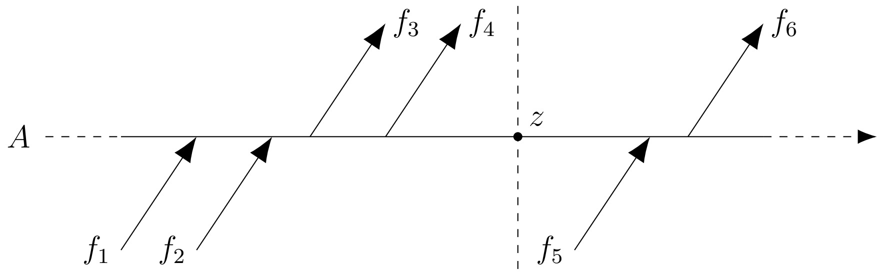


Figure 1: Visual representation of a *zero-instant.*

3 The Chunk Approach

As previously stated, this project means to build upon the foundations laid by [1], where, as illustrated in Section 2.1, the accuracy degree of AMRP is progressively increased in terms of time-windows of flight connections. From this point on, this solution shall be referred to as the *Time-window approach.* One particular observation on this approach concerns the lack of control over the amount of connections that might be added at each incremental step. That is, for a given flight, a time-window of one hour might include 30 as well as 0 new connections. This is particularly common in the case of instances which describe a *hub-and-spoke*[1] airline network: flights landing at hub airports will be able to take considerably more connections than flights landing at spoke airports. Mind that this peculiarity of the Time-window approach should not be regarded as a flaw. In fact, it should rather be considered as a checkpoint from which one may work on diverging, alternative approaches. This is what the project aims to achieve by proposing what shall be addressed as the *chunk approach*.

After providing an overview of its mechanisms, this section presents the chunk approach based on its multi-shot ASP implementation. Given how naturally SABP (see Section 2.3) fits into the implementation, this section shall also include its integration in the chunk approach.

3.1 A preliminary overview

The main purpose of the chunk approach is to offer better control over the accuracy degree's increase of AMRP. To this end, flight connections are partitioned in arbitrarily sized *chunks*. The size of the chunks is expressed by a constant denoted as `chunk_size`.

Diving deeper into the partitioning process, consider a flight f together with its set of connected flights $\mathcal{C}(f)$. First, the set should be equipped with a total order, that is, chronologically with respect to the departure time of the flights. In the eventuality where two flights happen to depart simultaneously, a lexicographic order over their IDs is adopted. During the last step, chunks are built following such an order.

Observing the multi-shot structure of a logic program implementing the chunk approach, consider an AMRP instance $\langle \mathcal{F}, \mathcal{T}, \mathcal{M} \rangle$ with $f_{max} \in \mathcal{F}$ denoting the flight with the highest amount of connections. After the partitioning, the set $\mathcal{C}(f_{max})$ gives rise to:

$$k = \frac{|\mathcal{C}(f_{max})|}{\texttt{chunk\_size}} + 1 \text{ chunks.}$$

As a consequence, the logic program shall include a subprogram for each chunk, with `step(i)` being the subprogram considering only the (chronologically) first i chunks of connections. This leads to the structure having subprograms `step(i)` for $0 < i \leq k$.

3.2 The implementation

The overview provided in Section 3.1 comes in handy also as an outline for the details of the implementation. For the sake of clarity, the following illustration follows the outline as close as possible. So far the report has been describing the structure of the multi-shot ASP program as being solely composed by subprograms concerned with the

[1] *Hub-and-spoke networks* are structured in main *hub* airports and outlying *spoke* airports. Spoke airports are connected exclusively through hub airports [2].

different stages of incremental solving (*i.e.* `step` *subprograms*). This depiction is slightly inaccurate, as one should also take subprogram `base` into account. This is a small, but nonetheless necessary clarification, as subprogram `base` represents the main stage of the contributions introduced by this project.

Lastly, the reader shall be made aware that, as for the development of the implementation, it was deemed more natural to reason in terms of *backwards connections*. That is, given a departing flight f, its set of backwards connections $\mathcal{B}(f)$ includes all arrival flights connected to f. Reasoning backwards makes no relevant difference with respect to the point of view adopted so far.

Ordering connection sets Building up from the contents of Section 2.1, it is useful to also introduce an encoding for non-carry-in flights. As previously mentioned, carry-in flights are assigned a priori, therefore they should not be considered in the solving process. A flight $f \in \mathcal{F} \setminus \mathcal{F}_c$, is encoded with the atom `range(f,a,s,t,b).`

Listing 1 shows the part of the encoding concerned with the ordering. After specifying the value for constant `chunk_size`, the following fundamental predicates are defined:

- `arrive(B,T).` encodes the presence of a landing at airport `B` at instant `T`, such that `B` is an airport where there is at least one departure. This second condition allows to immediately ignore part of the flights with no available connection.

- `depart(A,S).` encodes the presence of a departure at airport `A` at instant `S`. Slightly counterintuitively, the same atom is used to encode the flights encoded by atom `arrive/2`. This is done for the sake of succinctness, later in the encoding.

- `index(F,B,T,I).` encodes simultaneous arriving flights together with an index `I`. Such indexes define their position in the lexicographic order based on their IDs. Notice how switching from forwards to backwards reasoning shifts the lexicographic order from departing flights to arriving flights.

Lines 12-13 exploit the functionalities of Python enabled clingo. That is, the actual ordering process is delegated to the embedded Python script of the logic program. This speeds up significantly the ordering process, which would otherwise require a costly grounding phase. For the Python code referred to in the following lines, the reader is referred to [8].

Line 12 defines predicate `gather/1`. Given a flight activity encoded by `depart(B,T).` the Python function `insert(B,T)` is called. The function *gathers* the timings of the flight activities in different arrays depending on the airport they refer to. Finally, `insert(B,T)` returns the string `B`, which is finally used as ground term for predicate `gather/1`.

Line 13 defines the predicate `next_time/3` in order to relate successive time-points of flight activities at a given airport. During grounding, given a *gathered* airport `A`, the Python function `order(A)` is called. The function sorts the time-points array of `A`, thus returning an array of all pairs of successive time-points. The rule is, then, grounded for each pair in the returned array.

```
#const chunk_size = 5.

depart(A,S) :- range(F,A,S,B,T).
depart(B,T) :- arrive(B,T).
depart(A)   :- depart(A,S).

arrive(F,B,T) :- flight(F,A,S,B,T), depart(B).
arrive(B,T)   :- arrive(F,B,T).
index(F,B,T,I):- arrive(F,B,T),
                     I = #count{G : arrive(G,B,T), G <= F}.

gather(@insert(B,T)) :- depart(B,T).
next_time(A,T,S) :- gather(A), (T,S) = @order(A).
```
Listing 1: Ordering the set of connections.

SABP: Aircraft counting The counter on ground aircraft is the key element in the workings of SABP. As for the solution here presented, its encoding is shown in Listing 2.

Lines 1 to 3 define predicate `number/3`. This predicate encodes the counter `N` on the balancing of arrivals and departures in airport `A` at instant `T`.

Predicate `ac_count/3` encodes the actual count of ground aircraft. Given an airport `B`, the count is initiated from its first flight activity (Lines 5-6). From there, the counter is defined at any instant in which a flight activity occurs. More specifically, at any further instant, the counter is defined recursively (Lines 7 to 9) calculating the number of ground aircraft from the antecedent relevant instant.

Finally, predicate `zero_instant/3` simply denotes the instants in which such counters hit value zero.

```
 1 number(A,T,N) :- depart(A,T),
 2                  N = #sum{ 1, F : arrive(F,A,T);
 3                           -1, F : range(F,A,T,B,S) }.
 4
 5 ac_count(B,T,N)   :- number(B,T,N),
 6                      T1>=T : flight(B,T1).
 7 ac_count(A,S,C+N) :- ac_count(A,T,C),
 8                      next_time(A,T,S),
 9                      number(A,S,N).
10
11 zero_instant(B,T) :- ac_count(B,T,0).
```
Listing 2: Encoding of the aircraft counter.

Distance of connection Given a flight f, the term *distance* of a (backwards) connected flight f' is meant to denote the position f' takes in the ordered set $\mathcal{B}(f)$. As shown in Listing 3, to encode such information, the predicate `distance/3` is defined. This definition relies on the auxiliary predicates `tmp_arrive/3`, `prev_flight/3` and `tmp_depart/3`. An illustration of these predicates is thus provided.

An atom `tmp_arrive(F,B,T2)` denotes the fact that, given a time-instant `T2`, flight `F` is the latest arrival at airport `B`, previous to `T2`. Flight `F` is found through a backwards iteration which proceeds utilizing `next_time/3` atoms which, mind, do not distinguish between arrivals and departures. To this end, the rule in Lines 2-3 restricts the backwards iteration to proceed only if the time-instant just found is i) not an arrival; ii) not a zero-instant. When an arrival is found, the rule at Line 1 comes into play. Again, one must verify the second condition for the arrival's time-instant. The reason being that the arrival might coincide with a departure at a zero-instant. This is quite a rare eventuality, though, it would require the two simultaneous activities to be routed together, thus disallowing any connection between the arrival and later departures. With this in mind, one might already get an idea of how, by just adding or removing the second condition, the effects of SABP may be, respectively, activated or deactivated.

The predicate `prev_flight/3` is used to set an ordering on arrival flights at a given airport. The rule in Line 7 manages simultaneous arrivals: in this case one shall only increment index `I` to get the next arrival in the lexicographic order.

Similarly to `tmp_arrive/3`, predicate `tmp_depart/3` also performs a backwards iteration through flight activities. In this case, the iteration starts from a departing flight (Line 9) and proceeds until an arrival at the same airport is found. More specifically, the recursive definition at Line 10 is structured so that the latest arrival is also included by the relation denoted by the predicate. That is, given a departure at airport `A` at time-instant `S`, the smallest `T`, such that `tmp_depart(A,S,T)` holds, represents the time-instant of the latest arrival.

Everything is now set to observe the recursive definition of predicate `distance/3`. The rule at Lines 13-14 defines the base case (`distance = 0`) which concerns the *closest* (backward) connection. To this end, the aforementioned predicate `tmp_depart/3` is exploited. Since `tmp_depart/3` only deals with time-instants, predicate `index/4` is needed to obtain the first arrival in order, in the eventuality of simultaneous arrivals.

As for the inductive step, the focus shall be on Line 15. Given a connection at distance `N-1`, the connection at distance `N` is given by the precedent arrival at that same airport. More formally, given a departure f at time s from airport a, if f is (backwards) connected with arrival flight h landing at instant t, it shall be (backwards) connected with any flight g landing at a at $t' \leq t$.

```
1 tmp_arrive(F,B,T2) :- next_time(B,T1,T2), index(F,B,T1,1),
2                       not zero_instant(B,T1).
3 tmp_arrive(F,B,T2) :- next_time(B,T1,T2), tmp_arrive(F,B,T1),
4                       not zero_instant(B,T1), not arrive(B,T1).
5
```

```
prev_flight(F1,F2,B) :- tmp_arrive(F1,B,T), index(F2,B,T,I),
                        not index(_,B,T,I+1).
prev_flight(F1,F2,B) :- index(F1,B,T,I+1), index(F2,B,T,I).

tmp_depart(A,S,S) :- range(F,A,S,B,T).
tmp_depart(A,S,T) :- tmp_depart(A,S,X), next_time(A,T,X),
                     not arrive(A,X).

distance(F,G,0) :- range(F,A,S,B,T), tmp_depart(A,S,X),
                   index(G,A,X,1).
distance(F,G,N) :- distance(F,H,N-1), prev_flight(G,H,B).
```
Listing 3: Encoding for distances of flight connections.

In order to better understand the mechanisms of these predicates, a visual example is given in Figure 2, showing the backwards iteration through the timetable of an airport A. Dashed lines are used to highlight the iteration. Notice how consecutive departing flights f_4, f_5, f_6 converge to their first available connection f_3. This part of the iteration is performed by predicate `tmp_depart/3`. Once the first connection has been found, the iteration is resumed by predicate `distance/3`. These iterations are denoted by dashed lines with numbers indicating the distance's value. Another relevant observation concerns flights f_1, f_2. Being simultaneous arrivals, they are to be sorted lexicographically. Finally, notice how the backwards iteration starting from f_8 stops at the arrival of f_7. This is caused by the behavior of predicate `tmp_arrive/3` with respect to the zero-instant z. More specifically, there is no atom of the form `prev_flight(f,f7,A)`, for any flight `f`.

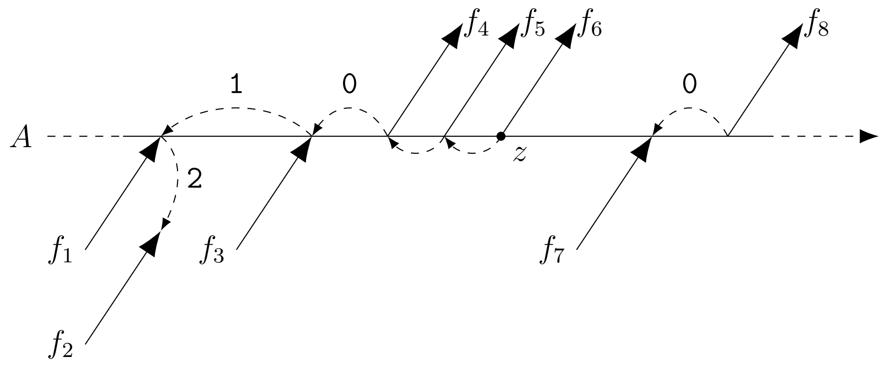


Figure 2: Visualization of connection distances of an airport's schedule.

Connection Encoding Finally, the information provided by these predicates needs to be further encoded to be consistent with AMRP as defined in Section 2.1. Whereas before a flight connection could be encoded through an atom `compatible/5`, in this case, additional information is required in order to specify the incremental step from which a connection should start being considered. To this end, the solver needs to know which chunk a connection belongs to.

With this in mind, predicate `compatible/6` is introduced. The predicate is implemented through the single rule shown in Listing 4. The body of the rule is rather intuitive: the distance of the connected flights is retrieved and the ground-time is calculated. Through the distance `N`, one can simply calculate the chunk of the connection by simply taking `N/chunk_size+1`. The result of this calculation consists exactly in the additional information the solver needs.

```
compatible(F1,B,T1,F2,G,N/chunk_size+1) :- distance(F2,F1,N),
                                           flight(F1,A,S1,B,T1),
                                           range(F2,B,S2,C,T2),
                                           G=S2-T1.
```

Listing 4: Encoding a connection with its chunk.

To build a clearer intuition on the contents of this paragraph, it may be useful to shift the focus from subprogram `base` to the incremental subprograms `step`. Although these subprograms are already well illustrated in [1], it may be useful to observe here how a subprogram `step` is restricted to a specific set of `compatible/6` atoms.

As mentioned in Section 2.2, a logic subprogram may be declared through the directive `#program`. Even more expressively, a declaration may include parameters, for example: `#program step(t).`, where the value for parameter `t` may be set from the embedded Python script. Mind that different values for `t` induce different subprograms.

Listing 5 shows how the directive is exploited in the case of AMRP. Given the subprogram `step(i)`, the rule in Line 2 is evaluated only through `compatible/6` atoms having `i` as their last argument, *i.e.* only considering connections from the i-th chunk.

```
1 #program step(t).
2 {route(F1, F2, G, t)} :- compatible(F1, B, T1, F2, G, t).
```

Listing 5: Beginning of a `step(t)` subprogram.

Here, one may notice an inconsistency with the formal descriptions given in Sections 2.2 and 3.1, where subprogram `step(i)` was said to deal with *all* chunks (or time-windows) j, such that $0 < j \leq i$. From the point of view of the implementation though, the incremental step i has subprograms `step(j)` with $0 < j \leq i$, each considering only the j-th chunk. However, since the subprograms are grounded and solved together, the final result is equivalent.

3.3 Relatively Sized (RS) Chunk Approach

As previously mentioned, the chunk approach emerged as an alternative to the Time-window approach. Similarly, the chunk approach was also taken as a lead to further develop another variant: the *relatively sized* (RS) *chunk approach.*

Observing different flights from an AMRP instance, one immediately notices a large difference in terms of the cardinality of their flight connection sets. Take for example two flights f, f', scheduled, respectively, at the beginning and at the end of the timetable: set $\mathcal{C}(f)$ is likely to be much larger than set $\mathcal{C}(f')$. With this observation in mind, one may notice how this reflects onto the solving process of the chunk approach as explained so far[2]. More specifically, this leads flights with fewer connections to consider the whole connection set very early in the incremental solving process and, consequently, to be unaffected by all later incremental steps.

The goal of the RS chunk approach is to mitigate this tendency to *"unfair"* incremental steps. It does so by, again, partitioning connection sets into chunks, albeit using a different integer constant to set their size: `chunk_rel_size`. This way, for a given flight, a chunk's size is defined relatively to the cardinality of its connection set. More formally, consider a flight f with connection set $\mathcal{C}(f)$. The RS chunk approach partitions $\mathcal{C}(f)$ into `chunk_rel_size` chunks, each containing:

$$\frac{|\mathcal{C}(f)|}{\texttt{chunk\_rel\_size}} \text{ flight connections.}$$

As it was the case for the AS chunk approach, the connections are grouped into chunks in chronological order.

Of course, the RS approach does not eliminate this tendency completely, as for large differences in terms of connection sets cardinality, there will always be unfair incremental steps. By contrast, one might argue that the tendency is not mitigated but merely distributed among different incremental steps.

Implementing the RS Chunk Approach Given the similarity with the AS chunk approach, this paragraph shall be rather brief. Again, the context is that of subprogram `base`. Any predicate, of which the definition is not shown, is to be interpreted as identical to its counterpart explained in Section 3.2.

First, constant `chunk_rel_size` is defined through the following directive:

```
#const chunk_rel_size = 70.
```

For a more intuitive understanding, the directive here shown would set chunks to each contain a seventieth of the flights' (backwards) connection sets.

The remaining additions/modifications concern the connections' encoding, here shown in Listing 6. Predicate `chunk_size/2` encodes the size of the chunks for a given flight. To this end, it retrieves the most distant (backwards) connected flight, adding 1 to its distance to get the cardinality of the connection set (Line 3). The Python function `chunk_size` simply delegates to the Python embedded script the previously mentioned calculations. Finally, predicate `compatible/6.` differs only in the fact that the chunk size needs to be retrieved from the `chunk_size` atom of the departing flight.

[2]When not clear from the context, this approach shall be referred to as the *absolutely sized* (AS) *chunk approach.*

```
chunk_size(F,@chunk_size(TOT+1,chunk_rel_size)) :-
                            range(F,A,S,T,B),
                            TOT=#max{ N : distance(F,_,N) }.

compatible(F1,B,T1,F2,G,N/CS+1) :- distance(F2,F1,N),
                                    flight(F1,A,S1,B,T1),
                                    range(F2,B,S2,C,T2),
                                    chunk_size(F2,CS),
                                    G=S2-T1.
```
Listing 6: Implementation of the RS chunk approach.

4 Experimental Results

The previous sections have showcased a variety of solutions and approaches that allow to tackle AMRP through Answer Set Programming. Combining all these factors together gives rise to a handful of settings. This section shall provide an exhaustive and critical analysis over these settings, describing the experimental results concerning the ones deemed as the most representative. Decisions that were taken over any specific setting shall be made explicit and justified.

When not otherwise specified, the settings in the following experiments shall be assumed to implement SABP and to run using clingo's parallel mode on an 8-threads competition. The tests were performed against the following three sets of randomly generated instances:

- **20 Small instances.** $20 \leq X \sim \mathcal{N}(50, 10) \leq 80$ flights per aircraft;
- **20 Medium instances.** $50 \leq X \sim \mathcal{N}(100, 20) \leq 150$ flights per aircraft;
- **20 Large instances.** $70 \leq X \sim \mathcal{N}(130, 25) \leq 190$ flights per aircraft,

where the notation $min \leq X \sim \mathcal{N}(\mu, \sigma^2) \leq max$ describes the truncated normal distribution over the number of flights to be assigned to each aircraft. Apart from the parameters just described, the settings used to generate the instances are the same as in [1]. To allow repetition of the experiments, the instances here used are all available in [8, `./exp_results/instances`]. The results here shown are to be understood as averages over runs against the instances considered in the particular experiment.

On a last note, all experiments were performed with clingo 5.4.0, with each solution's run limited to a 3660 seconds timeout. The runs were performed on a machine with Ubuntu 20.04 installed, an 8-core AMD EPYC-Rome CPU and a 16 GB RAM.

4.1 The effect of chunks' sizes on performance

The solving processes of both the AS and RS chunk approaches are, respectively, shaped by the values of constants `chunk_size` and `chunk_rel_size`. Table 1 provides the testing

	Absolutely sized chunks				Relatively sized chunks				Benchmark
`#const`	5	10	15	20	2	50	60	70	-
Time (s)	572.27	**542.33**	561.83	605.55	539.25	503.68	**432.60**	551.63	-
Cost	6418.55	6453.9	6423.60	**6398.35**	6443.80	6524.60	6443.80	**6408.45**	8761.75

Table 1: The effect of different constant values over the AS and RS chunk approaches.

results concerning runs over the set of small instances with different constant values. The data shows how different values affect time and cost (solution quality) performance. For the time being, the focus shall mainly stay on cost performance, as it provides a more accurate measure. Lastly, note that the values for the constants here considered were chosen arbitrarily, as a more rigorous tuning was deemed to be out of this project's scope.

The results do not highlight any relevant difference among the considered settings. However, all settings are shown to produce higher quality solutions than the benchmark value (computed at instance generation), allowing the conclusion that these are indeed promising settings. Perhaps, the most interesting entry concerns the RS chunk approach with `chunk_rel_size = 2`. This setting decomposes an AMRP instance into two subproblems: one considering half the connections; one considering all the connections. Such setting might appear futile in practice, as it barely exploits the multi-shot paradigm. Though, by closely examining the behavior of the two solving steps, relevant insights arise: on all tested instances, data (see [8, `./exp_results/results_small`]) shows the first subproblem to be unsatisfiable. As a consequence, the entire solving process is solely conducted by the second step, which, one should note, takes on the entire AMRP instance. The setting becomes, thus, equivalent to the single-shot one, with the only addition of the incremental step criteria (60 seconds timer per iteration; limit of three iterations without solution). The relevance of this result concerns the bound on the number of connections that may be ignored, that is, half of them before obtaining an infeasible subproblem. Obviously, this result is specific for the given instance size, though additional tests should be put into place, with more fine-grained incremental steps, to try and refine such bound. This information might, ultimately, help in tailoring the incremental solving process around the given problem size.

On a different note, the results from this first experiment provide benchmark settings for the AS and RS chunk approaches, namely, `chunk_size = 20` and `chunk_rel_size = 70`. From now on, results concerning these two encodings shall only consider these settings.

4.2 The impact of SABP

Solving optimization was initially introduced as a mere side-goal of this project. Nonetheless, it is only fair to recognize the impact SABP has had on its development. This preprocessing technique was previously said to cut significantly the amount of connections available in an airline schedule. Table 2 gives a clearer idea of the magnitude of its effect on instances of different sizes. The efficacy of SABP is immediately evident, with all instances being reduced to about a tenth of their initially available connections. More interestingly, the data underlines a trend: the larger the connection sets are, the more effective SABP is. Undoubtedly, this trend largely depends on the kind of airline network being considered: as for *hub-and-spoke* networks, similar preprocessing techniques are deemed successful and are often adopted in real-world scenarios [2].

Completing the picture, Table 3 shows the consequential effect SABP has on solving performances. For the time being, the absolute comparison between the solutions' performances is postponed, as it requires a more critical analysis (see Section 4.3). Instead, the focus should be on comparing SABP's effectiveness among different solutions. Unsurprisingly, SABP is shown to have a positive impact on performance, regardless of the encoding taken into consideration. Removing connections in a considerable number, SABP drastically simplifies instances, reducing their combinatorial complexity. For this reason, one would expect SABP to have a greater impact on the single-shot encoding rather than the multi-shot ones, as subproblems should enjoy a much weaker benefit from SABP compared to the entire instance. However, results concerning small instances show the opposite result. That may be justified by the exponentially growing difficulty in finding solution improvements over time. As for this case, reaching already a relatively low cost *without* SABP, the single-shot encoding has a harder time improving its performance once enhanced with SABP.

Moving on to the medium instances, the thesis in support of SABP is further reinforced. To avoid a prolonged testing phase, encodings not implementing SABP were tested only against a sample of 5 medium instances. Out of the 20 solving runs performed, only 3 were able to find any solution within the 3660 seconds timer limit. In contrast, encodings implementing SABP were successful in solving all 20 instances. Moreover, the average solution quality achieved is, in all cases, shown to be comfortably within the bounds established by the benchmark cost (Table 2).

Due to the unpromising results achieved by the encodings on medium instances when deprived of SABP, these settings were not tested on larger instances. On the other hand, approaches adopting SABP are shown to find solutions well within time-bounds. On average, the single-shot encoding is shown to be the only one satisfying the threshold of *"good solution"* provided by the benchmark cost. As for the other multi-shot encodings, the average results are deeply affected by outliers (of which the behavior is illustrated in Section 4.3). However, only in the case of the Time-window approach can outliers be deemed as the cause for a high average result.

Unfortunately, the data here provided hardly allows for any empirical conclusion

Instances	**No SABP**	**SABP**	**Removed Conns. (%)**	**Benchmark cost**
Small	28200	3422	87.8	8761.75
Medium	113744	7782	93.1	17185.20
Large	195858	10895	94.4	22629.05

Table 2: The effect of SABP over AMRP instances of different sizes.

on which setting best benefits from SABP. To this end, tests should be performed on instances bearing sizes in between small and medium. Having said that, more considerations on this comparison shall be provided in Section 4.3.

In conclusion, an additional observation concerns a possible synergy between the RS chunk approach and SABP. Among multi-shot solutions, results over small instances show SABP to be more effective when implemented with the RS chunk approach. A possible justification for this behavior might lie in the relative nature of the values used to set chunks' sizes. As previously shown, SABP drastically reduces the cardinalities of the connection sets. While the RS chunk approach *relatively* adjusts to the new cardinalities, the AS chunk and Time-window approaches do not adapt their incremental steps to the simplified AMRP instance. Proving whether this peculiarity of the RS chunk approach has any impact on performance requires further testing. To this end, it might be useful to also consider a *Relatively Sized Time-window approach*.

	AS chunks	**RS chunks**	**Time-window**	**Single-shot**	
`#const`	20	70	60	-	
	Small Instances				
Time (s)	735.72	**442.04**	460.76	3608.66	**No SABP**
Cost	6686.20	7064.95	6767.00	**6302.40**	
Time (s)	605.55	551.63	**502.92**	3606.90	**SABP**
Cost	6398.35	6408.45	6373.10	**6125.65**	
	Medium Instances				
Time (s)	2334.50[1]	≥ 3660	497.27[2]	≥ 3660	**No SABP**
Cost	31879.50[1]	No solution	16320.00[2]	No solution	
Time (s)	558.62	592.04	**464.22**	3617.67	**SABP**
Cost	15720.65	15670.15	15458.05	**14988.40**	
	Large Instances				
Time (s)	Not tested.				**No SABP**
Cost					
Time (s)	**847.06**	1039.71	935.93	3625.46	**SABP**
Cost[3]	49192.15	41446.95	28311.70	**22044.55**	

[1] Non-optimal solutions were found for two out of the five tested instances. The result here shown is to be understood of an average of the two solved instances;

[2] The result here shown represents the only solved instance out the five tested ones;

[3] *"Good solutions"* out of the 20 instances: 7 AS chunks; 7 RS chunks; 14 Time-window; 15 Single-shot.

Table 3: The effect of SABP on solution quality.

4.3 Setting the chunk approach to comparison

As previously anticipated, it is now time to re-observe the results provided by Table 3 for the sake of a performance comparison among the different encodings. Focussing on the multi-shot ones, the Time-window approach is shown to outperform the two chunks variants, with its advantage seemingly growing over instances of greater size. On a closer inspection, data concerning large instances (see [8, `./exp_results/results_large`]) shows the multi-shot encodings to produce rare but impactful outlying results with very high solution costs. This behavior is particularly critical in the case of the two chunks variants, thus causing a large increase on the average results. Nonetheless, the two encodings are able to produce these results within competitive solving times. This peculiar behavior is caused by the incremental steps criteria: the two encodings do indeed find a solution but are unable to improve it over the next three iterations. To counter this phenomenon, one could either increase chunks' sizes or loosen the criteria. The results displayed in Table 4 were obtained choosing the latter option (against a sample of 10 large instances), that is, increasing the limit of idle iterations (with no solution) to 5. Data shows a slight improvement in favor of the two chunks variants, with outlying results being better distributed among the three multi-shot encodings. Nonetheless, the Time-window approach holds on to its advantage.

Reconsidering the conjecture made at the end of Section 4.2, these last observations seem to raise a contradiction. According to the conjecture, one would expect the RS chunk approach to present better scalability than other encodings, as it should not require any re-tuning when dealing with larger instances. On the other hand, it ends up being increasingly outperformed by an encoding adopting time-windows of absolute size. Reiterating a previous statement, a more accurate tuning of constant `chunk_rel_size` might allow for more accurate conclusions on these results. In the opposite way, the scalability of the Time-window approach is rather surprising. On growing instances, the two chunks variants should present no surprises: each iterative step adds up a fixed (or relatively-fixed) amount of connections. This is not the case of the Time-window approach, as its 60 minutes windows should eventually end up being too small to affect the solving process, increasing the likelihood to breach the 3 idle iterations limit.

Moving towards an overall comparison, single-shot unequivocally claims supremacy

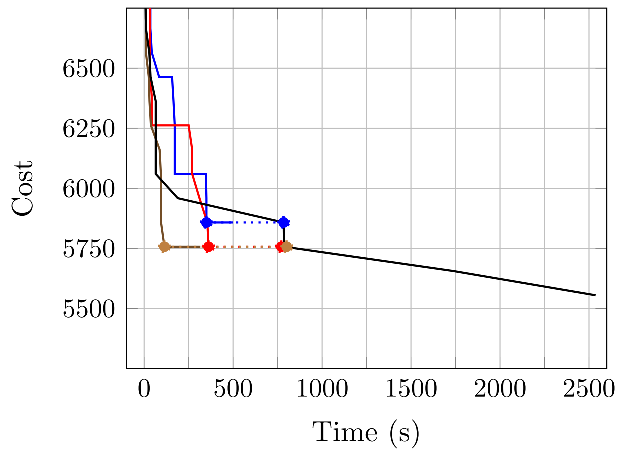
	AS chunks	**RS chunks**	**Time-window**	**Single-shot**
`#const`	20	70	60	-
Time (s)	1343.03	1316.41	**1106.05**	3623.91
Cost[1]	36049.8	34996.8	30379.7	**20978.7**

[1] *"Good solutions"* out of the sample of 10 instances: 7 AS chunks; 6 RS chunks; 8 Time-window, 10 Single-shot.

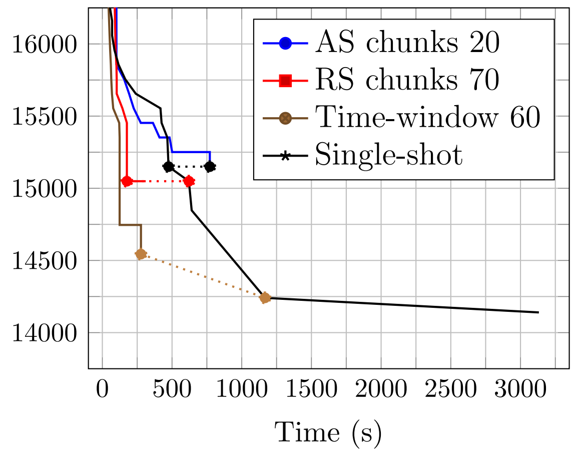
Table 4: Multi-shot encodings on a sample of 10 large instances (5 possible consecutive idle iterations).

in terms of cost performance. However, this claim needs to be put into perspective. To begin with, time performance can no longer be disregarded: while multi-shot encodings can, in the worst case, take about 15 minutes to solve an instance, single-shot regularly requires about an hour.

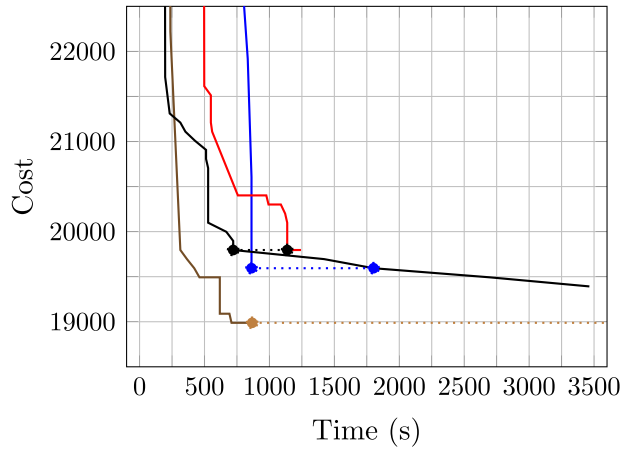
Apart from favoring multi-shot encodings, taking solving-time into account also allows for a more critical analysis. Analogously to the format adopted in [1, p. 11], Figure 3 displays four plots, each representing solution quality in function of solving time. The data concerns the runs of the four encodings, respectively, on small Instance 4 (3a), medium Instance 15 (3b) and large Instance 4. As for the latter instance, both results with 3 (3c) and 5 (3d) possible consecutive idle iterations are shown. The small and medium instances were chosen because deemed representative of the average case. As for large instances, Instance 4 was the only one not giving rise to any outlying results and was,



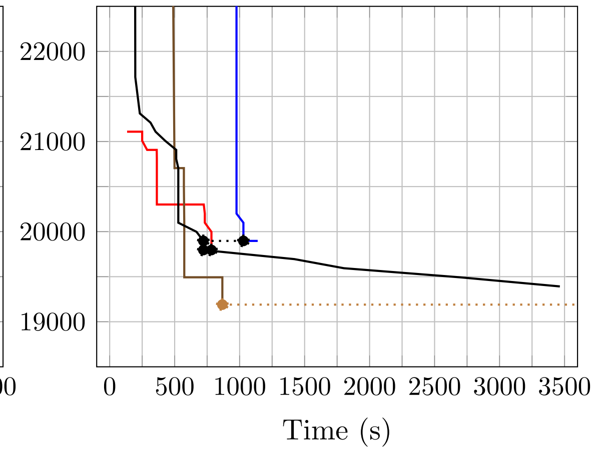
(a) Small Instance 4.



(b) Medium Instance 15.



(c) Large Instance 4.



(d) Large Instance 4 (5 idle iterations).

Figure 3: Cost performance of the encodings in time on differently sized instances.

thus, chosen by necessity. The plots make use of markers and dotted lines to highlight the time-difference required by single-shot to equalize the best solution found by the multi-shot encodings. On small instances, results show single-shot to require around 7 minutes to equalize the solution quality reached by the two chunks variants. On growing instances however, the pattern is hardly recognizable, with both of the chunks variants losing at least once their entire advantage to single-shot. In contrast, the Time-window encoding establishes an 11 minutes advantage on small instances and, more remarkably, is able to further extend the advantage on growing instances, to the point where single-shot is never able to equalize its solution quality. Briefly commenting Figure 3d, the plot highlights how, despite the less strict incremental step criteria, the two chunks variants still struggle to build even a temporary advantage over the single-shot encoding. Obviously, the data here presented hardly has any statistical significance, though it allows for an intuitive understanding of the encodings' solving process.

In [1], Tassel et al. were able to show multi-shot to have a much clearer advantage on single-shot. While this is indeed the case, the same conclusion does not quite extend to this report, as the results here shown are *"undermined"* by SABP. By decomposing a problem of large combinatorial complexity, multi-shot definitely provides an advantage in terms of time performance, with the advantage growing bigger and extending to cost performance on larger instances. Though, SABP was shown to follow the exact same behavior (Table 2). Thus, being SABP more impactful on an entire AMRP instance rather than its subproblems, the advantage of multi-shot over single-shot is mitigated. Eventually, SABP's advantage shall find its limit (in terms of percentage of connection removals), with instances further growing. On the other hand, multi-shot will keep extending the advantage. For this reason, to empirically show multi-shot's advantage over single-shot, even larger instances should be considered and, as a consequence, incremental step criteria appropriately adjusted.

4.4 Thread competition in parallel solving

This section shall end reporting on some experimental results that concern statistics of a slightly different nature than the ones presented so far. At the beginning of this section, parallel solving was defined as a default setting for the experiments to follow. Table 5 justifies this decision, showing the supremacy of 8-thread over single-thread solving. These

	AS chunks	**RS chunks**	**Time-window**	**Single-shot**	**Threads**
`#const`	20	70	60	-	
Time (s)	605.55	551.63	**502.92**	3606.90	8
Cost	6398.35	6408.45	6373.10	**6125.65**	(compete)
Time (s)	888.53	916.82	**759.94**	3605.17	Single
Cost	6714.25	6675.35	6660.70	**6256.95**	

Table 5: Single-thread and 8-thread (compete) solving in comparison.

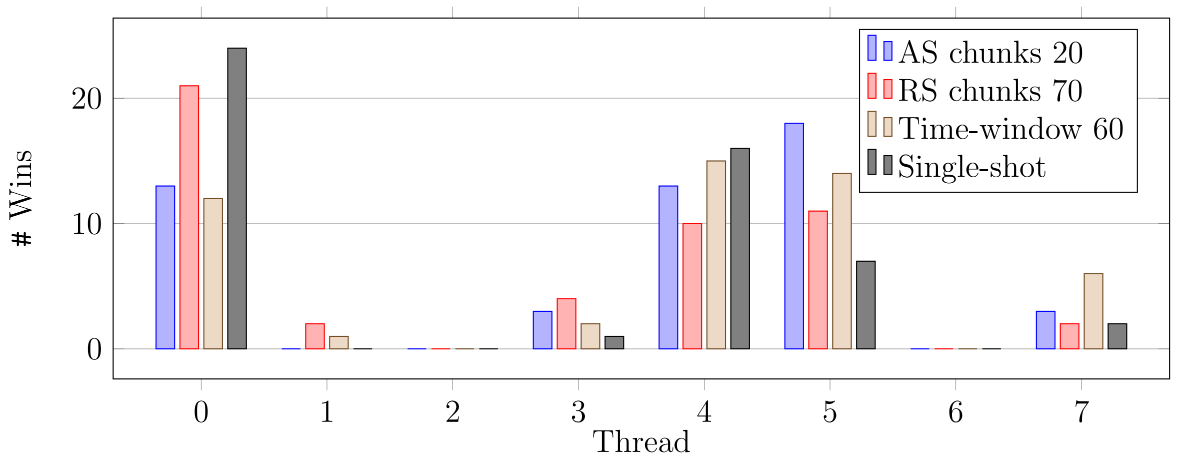


Figure 4: Winning threads in parallel solving for each specific encoding.

results were obtained through tests against small instances. The advantage of clingo's parallel mode lies in the possibility of solving an instance through different threads, each operating on a different configuration. When observing the cost performances achieved by different encodings, many runs were found to reach the same solution quality. This observation led to a further analysis, with the final aim of verifying possible correlations between winning threads and cost performances.

To begin the analysis, Figure 4 shows a histogram displaying the winning threads for each encoding over different runs. The tests were performed on a sample of 5 small instances, each being run 10 times. Data shows all encodings seemingly agreeing on the most winning threads: Threads 0, 4, 5. Analyzing all the parameters of their configurations is likely to open a too complex chapter. On the other hand, it is interesting to observe the specific decision heuristics adopted by the threads. Threads 0, 4 and 5 all adopt Variable State Independent Decaying Sum (VSIDS) [11]. These are not the only threads adopting this decision heuristic, therefore, it is hard to identify it as the reason of their success. In contrast, it is curious to observe how threads with no wins, *i.e.* Threads 2 and 6, are all and only the ones adopting heuristic BerkMin [12].

Concluding, the analysis over possible correlations between winning threads and solution quality is performed through the following case study. To this end, small Instance 5 was randomly selected. Figure 5 displays a jitter plot highlighting the solution quality achieved with respect to the winning thread. Given the dispersion of the marks along the Cost axis, one can hardly find any correlation between the two values taken into account. The only peculiar situation is described by Thread 5 running the time-window encoding, where four different computations are shown to achieve the exact same result.

Observing similar, if not equal results in terms of cost performance among different encodings should not be deemed an uncommon event. These similarities are, in fact, supported by the discrete and specific values that are used to compute the cost of a given

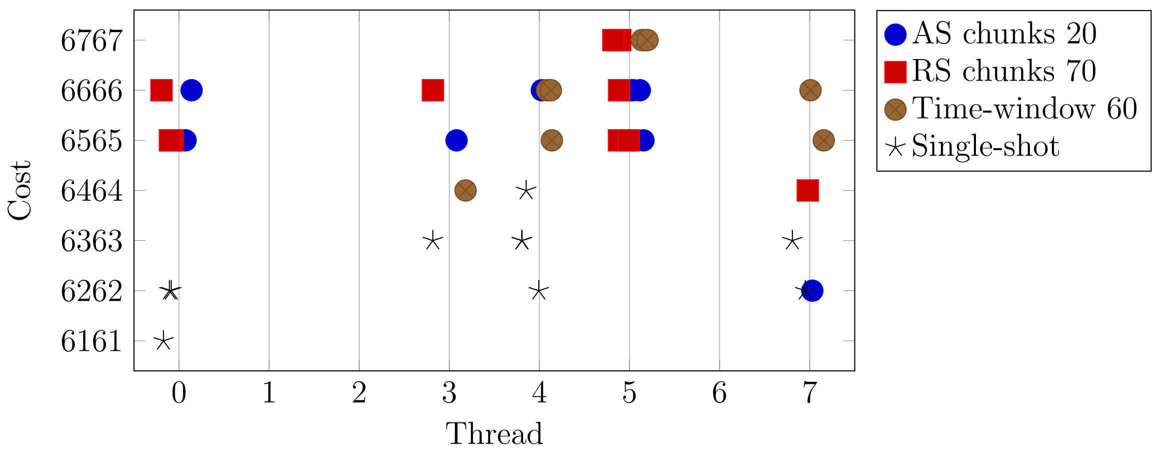


Figure 5: Solution quality achieved by different threads on small Instance 5.

solution [1, Section 3.3]. Therefore, it is possible to conclude that winning threads hardly have any correlation with these occurrences.

5 Conclusions and further work

Taking the work done in [1] as a starting point, the aim of this project was to propose alternative ways to exploit multi-shot ASP solving to tackle the Aircraft Maintenance Routing Problem (AMRP). The proposal takes shape into the so-called *chunk approach*, which further develops into two different variants: the *Absolutely Sized* (AS) chunk approach and the *Relatively Sized* (RS) chunk approach. Analogously to the time-windows adopted by Tassel et al., both variants decompose AMRP into an array of progressively harder subproblems, where the difficulty of a subproblem is determined by the amount of flight connections taken into account.

Experimental results show the chunk approach to slightly suffer the comparison against the Time-window approach. However, the main goal of the chunk approach was never to improve solving performance, as much as to offer better control over multi-shot's incremental steps. Both of its variants achieve this through specific hyperparameters that allow to arbitrarily set the amount of new connections to be added at each step.

Apart from developing new multi-shot ASP solutions for AMRP, further developments might focus on tweaking the already available approaches in different ways. To begin with, better results may be achieved through finer hyperparameter tuning. The tuning process could not only concern approach specific constants (*i.e.* `chunk_size` and `chunk_rel_size`) but also the incremental step criteria. Focussing on the RS chunk approach, Section 4.2 hinted to the possible development of a Time-window counterpart. The encoding would possibly work over time-windows sized according to the time-spans of the given connection sets. This solution might inherit both Time-window's better performances and RS chunk's (alleged) synergy with SABP. Furthermore, it may be interesting to try altering the order in which connection chunks become available during incremental solving. Currently, chunks (and time-windows) strictly join the solving process chronologically, following a *first-in-first-out* approach oriented towards favoring connections with lower ground-times. Lastly, one should not forget that the encodings were only tested on randomly generated instances. It might be interesting to extend the tests to real-world instances and verify whether there are approaches that fit best with specific kinds of airline networks.

Although not through the chunk approach, remarkable performance improvements were achieved implementing Simple Aircraft Balancing Preprocessing (SABP) [2, 10]. SABP is a preprocessing technique able to drastically reduce the amount of flight connections of a given AMRP instance. Apart from being very effective, it was also found to fit rather naturally into the ASP encodings here proposed. Given the way it affects an instance, the impact of SABP is more evident on single-shot solutions rather than multi-shot ones. Tests should be put into place to more accurately quantify this impact. Further developments might continue following the work done in [2], where *Consistency-based preprocessing* was proposed. This preprocessing technique was tailored around the constraint programming paradigm and results to be more effective than SABP. Whether there might be an ASP counterpart of this technique is yet to be understood.

References

[1] Pierre Tassel, Martin Gebser, and Mohamed Rbaia. "An ASP Multi-Shot Encoding for the Aircraft Routing and Maintenance Planning Problem". In: *13th Workshop on Answer Set Programming and Other Computing Paradigms*. 2020.

[2] Mattias Grönkvist. "The tail assignment problem". PhD thesis. Chalmers University of Technology, 2005.

[3] Ram Gopalan and Kalyan T Talluri. "The aircraft maintenance routing problem". In: *Operations research* 46.2 (1998), pp. 260–271.

[4] Kaushik Roy and Claire J. Tomlin. "Solving the aircraft routing problem using network flow algorithms". In: *2007 American Control Conference*. 2007, pp. 3330–3335. DOI: 10.1109/ACC.2007.4282854.

[5] Lloyd Clarke et al. "The aircraft rotation problem". In: *Annals of Operations Research* 69 (Jan. 1997). DOI: 10.1023/A:1018945415148.

[6] Martin Gebser et al. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2012.

[7] Martin Gebser et al. "Multi-shot ASP solving with clingo". In: *Theory Pract. Log. Program.* 19.1 (2019), pp. 27–82. DOI: 10.1017/S1471068418000054. URL: https://doi.org/10.1017/S1471068418000054.

[8] *Aircraft Scheduling - Github Repository*. `https://github.com/Alex-Dell1/Aircraft_Scheduling`.

[9] Mohammed M. S. El-Kholany, Martin Gebser, and Konstantin Schekotihin. *Problem Decomposition and Multi-shot ASP Solving for Job-shop Scheduling*. 2022. DOI: `10.48550/ARXIV.2205.07537`. URL: `https://arxiv.org/abs/2205.07537`.

[10] Christopher A Hane et al. "The fleet assignment problem: Solving a large-scale integer program". In: *Mathematical Programming* 70.1 (1995), pp. 211–232.

[11] Matthew W Moskewicz et al. "Chaff: Engineering an efficient SAT solver". In: *Proceedings of the 38th annual Design Automation Conference*. 2001, pp. 530–535.

[12] Eugene Goldberg and Yakov Novikov. "BerkMin: A fast and robust Sat-solver". In: *Discrete Applied Mathematics* 155.12 (2007). SAT 2001, the Fourth International Symposium on the Theory and Applications of Satisfiability Testing, pp. 1549–1561. ISSN: 0166-218X. DOI: `https://doi.org/10.1016/j.dam.2006.10.007`. URL: `https://www.sciencedirect.com/science/article/pii/S0166218X06004616`.