A Constraint Programming Model for Tail Assignment

Mattias Grönkvist[1]

Carmen Systems AB
Odinsgatan 9
S-411 03 Göteborg, Sweden
mattias@carmensystems.com

Abstract. We describe a Constraint Programming model for the Tail Assignment problem in airline planning. Previous solution methods for this problem aim at optimality rather than obtaining a solution quickly, which is often a drawback in practice, where quickly obtaining solutions can be very important. We have developed constraints that use strong reachability propagation and tunneling to a column generation pricing problem to form a complete and flexible constraint model for Tail Assignment which is able to quickly find solutions. Results on real-world instances from a medium size airline are presented.

1 Introduction

Tail Assignment is a problem from the airline planning field. The problem consists in creating individual aircraft routes for a set of flights and ground activities, subject to a number of operational rules. The problem lies in the boundary region between planning and operations control, and it is therefore crucial to quickly be able to provide working solutions. In a longer perspective, e.g. when integrating the aircraft routes with crew planning, optimization also becomes important. For example, minimizing the number of aircraft swaps by flying crews can greatly improve the robustness of the complete solution in light of disruptions. Here, we will only be interested in finding a solution, ignoring the cost aspect altogether.

We have previously shown [11] how constraint programming can be used to improve the performance of a mathematical programming model for the Tail Assignment problem. One problem with the mathematical programming model, which is based on column generation, is that it is not directly suited to finding solutions quickly. It rather puts optimization in focus, and is therefore very suitable indeed for the longer-term planning, but less suitable when a quick solution is desired. It is possible to obtain initial solutions more quickly with this model by integrating it with constraint programming, and forcing the mathematical model to work very aggressively.

However, it would be desirable to have a method that is more aimed at finding solutions quickly. Such a method could be used alone, but also to provide the initial solution to the column generator for further improvement, and thus

make it possible to first provide an initial solutions very quickly, and then more optimized solutions after some more time. Finding a solution to Tail Assignment is a very hard problem, and practice has shown that greedy construction methods have problems finding solutions. We have therefore focused on constraint programming as a means to find initial solutions quickly. We have already had a partial constraint model before [14], to use for integration with the column generator. But this model is not able to handle some of the more complicated constraints, and can therefore not be used to find proper solutions. Attempts have been made to complete the model [6], but unfortunately the results were not that successful.

In this article we describe the development of new constraints and ordering heuristics to build a complete constraint model for Tail Assignment. Section 2 we start by introducing Tail Assignment, and in Section 3 we describe the basic constraint model, and some simple extensions of it. Sections 4, 5 and 6 describe the new constraints and ordering heuristics introduced to complete and solve the model, and Section 7 show computational results on real-world Tail Assignment instances. In Section 8 we conclude.

2 Tail Assignment

Tail Assignment is the problem of deciding which individual aircraft (identified by its *tail* number) should cover which flight. Each aircraft is thus assigned a *route* consisting of a sequence of flights, and possibly other activities such as maintenance, to perform. Tail Assignment deals with individual constraints, flights which are fixed in time, as well as individual rules for each tail. The planning period is typically one month. The purpose is to really create a solution that is possible to operate, satisfying all rules and regulations. The most basic rules are rules which only depend on two flights, so-called connection-based rules. For example, there must be a certain minimum buffer time between a landing and the next take-off.

Another important set of constraints are the *flight restriction* rules, which forbid certain aircraft to operate certain flights. There can be many reasons for the restriction – there can be a *curfew* for the arrival airport and some aircraft, because the aircraft violates noise or environmental restrictions. But there can also be more down-to-earth reasons, like the aircraft not having the required in-flight entertainment system or extra fuel tanks required for a long flight. Either way, the result is that an aircraft is restricted from operating a flight.

Finally, there are the maintenance rules. Aviation authorities require that all aircraft undergo various types of maintenance activities regularly. There are many maintenance types, depending om aircraft type, registration country, and airline. Typically, the rules specify that aircraft must undergo maintenance every X hours, or every Y landings. Airlines often also require that their aircraft return to a maintenance base frequently, even if no maintenance is done, to increase robustness in case disruptions occur. These rules typically specify that aircraft must come back to a maintenance base every Z days.

The normal representation of the Tail Assignment problem is in terms of a *flight network*. In the flight network, each node represents a flight, or some other activity such as a preassigned maintenance activity for specific aircraft, and each arc represents a connection between two flights or activities. For example, if operating flight f followed by flight f' is allowed according to connection rules, the connection from f to f' is considered *legal*, and the flight network will contain an arc between nodes f and f'. Since we are solving a dated problem, where flights are fixed in time, there are *carry-in* activities in the beginning of the period representing the last flights operated by each aircraft in the previous planning period, and the network is acyclic. The goal is now to find paths (routes) through the network for all aircraft, starting at the carry-in activities, such that all flight nodes are covered exactly once, and all rules are satisfied.

Variations of the Tail Assignment problem exist, for example the Aircraft Routing [9] and Aircraft Rotation [1, 5] Problems. [11] contains a more thorough review of the literature about Tail Assignment and similar problems. Problems similar to Tail Assignment have been subject to constraint programming research, for example the Crew Scheduling Problem [7, 12], and the Vehicle Routing Problem [2, 16].

3 The Basic Model

This model was first described by Kilborn in [14]. The model has three sets of variables. Let us call the set of flights F and the set of aircraft A. First, there are `successor` variables for all flights $f \in F$, containing the possible successors (legal connections) of the flight f, which we denote by S_f. Since the rules related to flight-to-flight connections are already modeled in the `successor` variables, no such constraints need to be explicitly added to the model. Then there are also `vehicle` variables for all flights, initially containing the aircraft in A that are allowed to operate the flight, A_f. These variables model preassigned activities as well as curfew rules.

Only two constraints are present. Firstly, since all flights must have unique successors to form disjoint routes through the network, all `successor` variables must take unique values. Therefore an `all_different` [15] constraint over all `successors` is added. Flight nodes which can represent route endings reconnect back to the carry-in activities. Secondly, to maintain consistency between the `successor` and `vehicle` variables, a special tunneling constraint is added. Observe that once completely instantiated, the `successor` and `vehicle` variables both describe the solution completely. The `successor` variables obviously give a direct route for an aircraft, and the `vehicle` variables specify which flights are assigned to a certain aircraft. Since the flights are fixed in time, simply sorting all flights assigned to an aircraft gives the route for this aircraft. The tunneling constraint is implemented as a single constraint that is propagated each time a variable is fixed, and takes appropriate action to keep the variables consistent:

```
    vehicle[f] == a ⇒ POST  element(successor[f],vehicle,a)
successor[f] == f' ⇒ POST  vehicle[f] = vehicle[f']
```

The constraints are thus not posted all at once, but rather on demand whenever a `vehicle` or `successor` variable is fixed. It could have been possible to propagate each time a variable is *changed*, rather than fixed, but for simplicity and effectiveness we only propagate when variables are fixed. Observe that while the expressions above state the necessary actions to keep the variable consistent, other things can be enforced as well, to accelerate the propagation. For example, it is possible to directly remove j from the domains of overlapping flights when `vehicle[i] == j`. The constraints will make these assignments impossible, but from a performance point of view it is often beneficial to remove them directly. The `element(b,A,c)` constraint forces the `b`th value in vector `A` to take value `c`, i.e. `A[b] = c`.

3.1 Ordering Heuristics

To make this model behave properly is to crucial to define good variable and value ordering heuristics. Only `successor` variables are instantiated. The reason is that these variables are propagated much more than the others, thanks to the strong consistency algorithm for `all_different` due to Régin [15]. The `successor` variables are fixed in order of increasing domain size, i.e. using the well-known *first-fail* ordering, with the exception of preassigned flights, which are fixed first. Values are chosen in increasing connection time order, except for long connections. If a connection is long, for example over night, we try to connect to the next preassigned activity first, rather than the next possible flight.

The result is a model that captures all Tail Assignment constraints except the maintenance rules. The model works very well for problems without many curfew rules, showing almost linear time behavior for problems of increasing size. For more results we refer the reader to [14]. Unfortunately the model does not work well at all in the presence of many curfew rules. The reason is simply that the propagation for these constraints, which are embedded in the `vehicle` variables, is very weak. Simply put, we cannot get the partial routes we create to fit together, because the aircraft are incompatible with them.

3.2 Improvements of the Basic Model

It is fairly straightforward to improve the basic model above with a few more variables and constraints. Observe that the improvements in this section are only added to improve propagation, and as an effect the solution speed, of the basic model. Improvements to make the model more complete will be the topic of Sections 4 and 5.

Firstly, we can observe that since flights are fixed in time, flights which overlap in time can never be operated by the same aircraft. When we fix the `vehicle` of some flight we can therefore remove the fixed vehicle from the domains of all overlapping flights, as mentioned above. Also, we can add `all_different` constraints over all flights which pass a certain point in time. Observe that adding an `all_different` for *all* flights overlapping another flight is not a good idea, as an aircraft operating a flight overlapping only the beginning of the flight might

be able to also operate a flight that overlaps only the end. The problem with adding `all_different` constraints is to decide where to add them. Since each `all_different` is only valid for one specific time, there is an infinite number of such constraints that could be added. Our strategy is to initially add one `all_different` for the start time of each flight which has some restriction, i.e. which cannot be operated by all aircraft:

$$\texttt{size(vehicle[f])} \,!= \texttt{size(A)} \Rightarrow \texttt{POST} \quad \texttt{all\_diff(successor[G(f)])}$$

$$\texttt{G(f)} \quad : \quad \text{Flights overlapping the start time of flight } \texttt{f}$$

Secondly, we add `predecessor` variables, representing the possible predecessor flights P_f of a flight. Considering predecessors explicitly has the benefit of finding flights that only has one predecessor, but whose predecessor has multiple successors. To keep the `successor` and `predecessor` variables consistent, we add an `inverse` constraint. The `inverse(f,f')` constraint, which has been implemented as a single constraint, simply ensures that `f` exists as a predecessor to `f'` if and only if `f'` exists as a successor to `f`:

`inverse(f,f'):`

`successor[f]` contains `f'` $\iff$ `predecessor[f']` contains `f`

We also include treatment for `predecessor` variables in the tunneling constraint, to keep it consistent with `vehicle` variables. These simple improvements have very positive effects on the model, decreasing the number of backtracks used and increasing the propagation.

4 Handling the Flight Restriction Rules

As we have already discussed, the flight restriction rules are in fact modeled already in the basic model, but when several restrictions are present, the performance of the search deteriorates. Since the propagation for the `vehicle` variables is poor, we end up generating partial routes, which do not fit together because of the restrictions. The effect is excessive backtracking, so-called *thrashing*. It is quite obvious that one way to deal with this problem is improved propagation for the `vehicle` variables. The introduction of `all_different` constraints over `vehicle` variables overlapping certain times in the previous section is an attempt at improving this, but it is not enough.

Instead, the key is to switch from the local 'successor-view' of the problem to a more route-oriented view. As our ultimate goal is to create routes for all aircraft, it seems like a good idea to introduce this point of view into the model. Since we want all flights to be operated by exactly one aircraft, we need to make sure at all times that each flight is contained in at least one route which satisfies the flight restrictions, for one of the aircraft. We do this by keeping track of the number of successor and predecessor flights that can be reached by each aircraft. That a flight can be reached by aircraft a means that a route from the carry-in of a to the flight exists. A flight can be reached in the forward direction, via

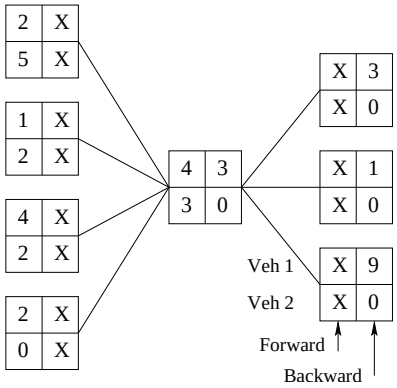

Fig. 1. An example of forward and backward labels with only two vehicles. Labels depending on flights not in the figure have been replaced by X.

`successors`, and in the backward direction, via `predecessors`. The number of reachable neighbors is relatively cheap to update, and gives the information that we need. If no successor of a flight f can be reached by aircraft a, flight f itself can obviously not be reached, and thus not operated, by aircraft a. Similarly, if no predecessor can be reached by a, neither can f. But if at least one predecessor and one successor can be reached by a, and $a \in A_f$, then f can be operated by a.

For each flight, we maintain two *labels* for each aircraft. One label, called the *forward label*, counts how many predecessors have a non-zero forward label for the aircraft, i.e. how many predecessors can be reached, in the forward direction, by the aircraft. The other (the *backward label*) counts how many successor flights have a non-zero backward label for the vehicle. Let us call the forward label of flight f for aircraft a fl_f^a and the backward label bl_f^a. Let us further denote the set of carry-in flight by F_c, and let c_f^a be 1 if f is the carry-in flight of aircraft a and 0 otherwise. Now, the labels have the following relationship:

$$fl_f^a = c_f^a \quad \forall f \in F_c, \forall a \in A \tag{1}$$

$$bl_f^a = c_f^a \quad \forall f \in F_c, \forall a \in A \tag{2}$$

$$fl_f^a = \sum_{f' \in P_f \& fl_{f'}^a > 0} 1 \quad \forall f \notin F_c, \forall a \in A \tag{3}$$

$$bl_f^a = \sum_{f' \in S_f \& bl_{f'}^a > 0} 1 \quad \forall f \notin F_c, \forall a \in A \tag{4}$$

In the example in Figure 1, all of the predecessors have non-zero forward labels for vehicle 1, and the forward label is thus 4. Since none of the successors have non-zero backward labels for vehicle 2, the backward label is 0, and so on. As a consequence, the flight cannot be covered by vehicle 2.

4.1 Maintaining the Labels During Search

To initially set the forward and backward labels to their correct values is simple: Just set the forward labels by counting labels for predecessors, starting with the carry-ins and ending with the flight with the latest departure time. The backward labels are set in the opposite direction. The carry-in activities always have one forward and one backward label, and aircraft restricted from flying a flight of course always get forward and backward labels 0. More complicated is maintaining the correct labels once `successor`, `predecessor` and `vehicle` domains shrink, and in case backtracking occurs, increases in size.

Let us first assume that f' is removed from `successor(f)`. In case f' is a carry-in activity, nothing happens, as we want to maintain one single label on carry-ins. If not, apply the following algorithm (in C-style pseudocode):

```
successor_removal(f, f')
begin
  for each aircraft a
    if (forward_label[f,a] == 0 OR
        forward_label[f',a] == 0)
      do nothing
    else
      --forward_label[f',a]
      if (forward_label[f',a] == 0)
        forward_remove_aircraft(a, f')
    end if
  end
end

forward_remove_aircraft(a, f')
begin
  q = empty queue
  q.push(f')
  while (q not empty)
    i = q.pop()
    for (all successors j of i)
      if (forward_label[j,a] != 0 AND
          j is not carry-in flight)
        --forward_label[j,a]
        if (forward_label[j,a] == 0)
          remove a from vehicle(j)
          q.push(j)
        end if
      end if
    end
  end
end
```

What happens is that we first decrease the forward label on flight f' by 1 if both f and f' have non-zero forward labels, as one of the routes to flight f' is now removed. If this makes the forward label go to 0, this will potentially affect all successors of f', and we must call `forward_remove_aircraft()`. This function iteratively decreases forward labels as long as some label becomes 0 when decreased. The use of a queue makes sure that we treat the labels in the proper order, i.e. we treat all predecessors of f before we treat f. Upon termination of `successor_removal`, all forward labels are correct with respect to the variables. Backward labels are treated analogously.

Now instead imagine that aircraft a is removed from flight f. The algorithm for this case is shown below, and should be self-explanatory in light of the discussion above.

```
aircraft_removal(a, f)
begin
  if (forward_label[f,a] > 0)
    forward_label[f,a] = 0
    forward_remove_aircraft(a, f)
  end if
  // And the same for the backward label
end
```

Value insertions are treated much the same way as the removals, except that instead of updating forward/backward when the label becomes 0, we update when the label is increased from 0 to 1. We call this propagation algorithm the *reachability algorithm.*

Looking at the complexity of the propagation algorithm, the worst case is when we are forced to update all labels for all flights each time a value is removed or inserted. If we have p aircraft, n flights and each flight has m successors in average, the worst-case complexity of each successor removal is thus $\mathcal{O}(pnm)$, which is not that good. However, in practice we will not experience this behavior at all. The reason is that in the initial stage of the search, most flights can be reached via several routes, so most labels will have high values. Labels will thus seldom become 0, which means we will only have to update one or a few flights at each removal. As the problem gets more fixed, more labels take values close to 0. But on the other hand, each flight does not have as many successors at this stage, which means we will still not have to update that many labels. So in practice, this propagation algorithm works well, as Section 7 will show.

It should be observed that this propagation algorithm could probably have been stated in terms of simpler constraints rather than as a 'global' propagation algorithm. The propagation algorithm does not provide extra propagation because of the fact that it is implemented as a global algorithm. The decision to model it this way is rather motivated by performance. Also, the algorithm has been implemented as a constraint in the sense that it reacts to domain changes as a constraint, but it really only consists of the propagation algorithm.

5 Handling the Maintenance Rules

As discussed in Section 2, maintenance constraints are expressed as 'each aircraft must return to a maintenance base every X flying hours/Y landings/Z days for maintenance taking T hours'. Observe that several such rules can apply simultaneously, as there are several types of maintenance, which might take different time to perform, and might not all be possible to perform at the same base. Here, we will only require that the aircraft are provided maintenance *opportunities* with the proper intervals, and we will assume that whenever there is an opportunity the aircraft will be maintained. In practice an aircraft returning to base e.g. 3 nights in a row might not be (fully) maintained every night. But as long as there are enough opportunities, it will be possible to maintain all aircraft enough.

Before describing the constraints to handle the maintenance rules, let us briefly review the column generation approach to Tail Assignment [10], as we will make use of it later. Column generation is a well-known mathematical programming technique [3, 7, 8, 13], used e.g. when the number of columns (variables) for a linear program are too many to enumerate. Instead, columns are generated dynamically, using dual cost information, until no more improving column exist. In the case of the Tail Assignment model, the columns are routes for individual aircraft. We have thus previously developed a module that given a set of dual costs can find a set[1] of minimum cost routes, satisfying all flight restriction and maintenance rules. In this module, which is called the column generation *pricer*, the maintenance rules are modeled as *resource constraints*, making the pricing problem a resource-constrained shortest path problem. The implementation follows roughly that described in [4].

Now, since we already have an implementation of the maintenance rules in the pricer module, and would like to avoid implementing the same thing twice, it seems reasonable to try to re-use the pricer for the constraint model as well. The pricer implementation is highly tuned and general, as it is user-customizable via the domain-specific Rave language [10], making it possible to model any kind of rule that can be modeled as a resource constraint, not only maintenance rules. Completely re-implementing this functionality in the CP model would be cumbersome indeed. However, as the pricer is labeling-based [4], it can easily allows us to check feasibility by checking whether

1. Each flight is reachable, i.e. there is at least one route arriving at each flight
2. There exists at least one legal route per aircraft

Unfortunately, this check is *incomplete*, in the sense that a partially fixed network can pass the feasibility check even if it is infeasible, e.g. if a flight is the only possible successor of several flights. However, for a fully fixed network it is complete, so we will never allow solutions violating the maintenance constraints.

[1] The least cost column is found, along with a predefined number of other low-cost columns, but not necessarily the least cost ones.

By creating a constraint that tunnels the changes in the variable domains to the internal structures used in the pricer, and does the above feasibility check, we have created a constraint that makes sure that a solution must be maintenance feasible. The constraint (which we call the *pricing constraint*) is not very strong in terms of propagation, but fortunately experience tells us that the maintenance rules are seldom very tight, so this is not be a huge problem in practice. The only propagation done by the constraint is that in case no route for a certain aircraft exists to a flight, this aircraft is removed from the `vehicle` domain.

One major concern with this constraint is that fact that it is fairly expensive to check. One of the drawbacks of using code 'unrelated' to the rest of our CP model is that it is difficult to customize it perfectly to our needs. We can customize it to a large extent by deciding on the number of labels, number of generated routes etc, but doing the check still takes too long for it to be done at every domain change. Instead, we have to settle for checking/propagating the constraint with a certain interval during the search, and at the very end. However, as the next section will show, this fits rather well into the full picture.

6 Ordering Heuristics

We have now described all necessary constraints to form the Tail Assignment model. However, as Section 7 will show, this is not enough unless well-working variable and value ordering heuristics are used. Unfortunately, the old variable ordering heuristic of instantiating preassigned activities first, and then fix according to increasing `successor` domain size, does not work well at all. Instead, we must again resort to routes. Instead of fixing single variables based on their local properties, e.g. domain size, we create an entire route, which we know satisfies all flight restrictions and maintenance rules, and let the route set the order in which the `successor` variables are instantiated. Once we have fixed all `successors` in a route, we create another route, and so on, until all aircraft have routes.

This goes hand-in-hand with the feasibility check performed in the pricing constraint, as this provides access to the necessary routes. When checking feasibility using the pricer, we do a labeling run which results in routes for all aircraft, if such exist. This suggests the following variable ordering strategy:

- Whenever all `successors` in a previous route have been fixed
  - Check feasibility of the pricing constraint
  - If infeasible, backtrack
  - If feasible, take one of the generated routes as the next route to fix

One question is which route to choose to fix. Our strategy is to fix the aircraft in order of decreasing number of flight restrictions. That is, to avoid them getting trapped when most of the network is fixed, we start by fixing routes for the aircraft which have the largest number of flight restrictions, i.e. the aircraft a for which $\sum_f f_a$, where f_a is 1 if flight f can be operated by aircraft a and 0 otherwise, is minimal. Also, to avoid the last few aircraft we fix from getting

stuck because there are no maintenance opportunities left, we want to use as few maintenance opportunities as possible each time we fix a route. This can be at least approximately achieved by setting the dual costs properly for the pricing constraint. Remember that the pricer finds a set of least-cost routes with respect to the provided duals[2]. Since long, typically overnight, connections are used to perform maintenance, we want to penalize the use of such connections. This is done by putting a large negative dual cost on using long connections. We therefore set the dual cost of all flights to the negation of the sum of the connection times to successors.

Finally, to speed up the search, in case we need to backtrack, we backtrack the entire route we are currently fixing instead of backtracking a single step. This makes the whole search incomplete, but is only done to avoid excessive backtracking, and excessive checking of the pricing constraint. If we find a conflict half-way through fixing a route, backtracking by single steps until we find a feasible node could potentially force us to check the pricing constraint far too many times to be efficient. So instead we backtrack all the way to the beginning of the route, where we know the pricing constraint has been checked. To avoid the same route from being re-generated, we add a penalty to the 'dual cost' of all flights we attempt to fix.

7 Computational Results

We have implemented the constraints and ordering heuristics as described above, and in this section we will present computational results for a set of real-world test instances. The instances come from different planning months at the same medium size airline. The aircraft included are a mix of M82, M83 and M88 aircraft. The underlying CSP solver is one designed in-house.

Table 1 presents the instances, and shows the running times of a fixing heuristic method [10] based on column generation, which was the best known method for quickly producing solutions prior to this work, and the constraint model. It should be observed that the performance of the fixing heuristic also relies heavily on constraint propagation, as it uses the basic CP model to check that fixes proposed by the fixing heuristic do not lead to conflicts. The pure column generation fixing heuristic is not included in the comparison, as it has the property that it can leave flights unassigned. This is even more likely to occur if the algorithm is tuned so as to compete with the running times of the approaches presented here. Comparing running times of a heuristic that can leave flight unassigned with heuristics that cannot do this simply does not make sense, and hence this comparison was skipped.

The running times in the table, which are all rounded to even integer seconds, include problem setup and some preprocessing and bound-calculation steps as presented in [10]. The number of flights reported is the number of activities given to the constraint model. Due to the preprocessing, each activity might consist

[2] In fact with respect to the reduced cost. But if real costs are set to 0, the optimization will only be over the duals.

Table 1. Comparison of solution times using a column generation-based fixing technique, and using the constraint model.

Instance	Flights	Aircraft	Rules	Fix. Heur. Time (s)	CP Time (s)
Instance 1	2388	33	1	140	70
Instance 2	2965	31	1	247	98
Instance 3	2757	31	1	184	82
Instance 4	2379	31	2	232	99
Instance 5	2839	31	1	175	78
Instance 6	2652	31	2	209	99
Instance 7	2474	31	1	156	68
Instance 8	2858	31	1	1468	106
Instance 9	2604	31	1	218	66
Instance 10	2040	30	3	321	112
Instance 11	2232	30	3	235	112
Instance 12	2226	30	3	249	134
Instance 13	2163	30	3	184	100

of several atomic flights (*legs*). The actual number of flight legs is therefore roughly twice the number reported in table 1. The 'Rules' column gives the number of maintenance rules that are present for each instance. For all instances, a rule specifying that the aircraft should return to base with regular intervals in present. For some instances, so-called 'A-check' and lubrication maintenance rules are also present. It is clear that the constraint model produces solutions significantly faster than the old method. Roughly speaking, the constraint model is about twice as fast.

Table 2 shows the benefit of the reachability propagation. The table shows the behavior of the basic model compared to the basic model with reachability propagation. When the reachability propagation is included, the variable order is changed by simply taking the first non-fixed flight from the most restricted aircraft first. We thus do not generate and fix entire routes, but still use the reachability labels to help guide the search. Entries marked with a star means that no solution was found within the predefined limits, which were set to maximum 1800 seconds or as many backtracks as there are flights in the problem. The measured times are in these cases the times to reach this many backtracks. The reachability propagation clearly helps, as without it only two instances can be solved within the limits, while when it is added, all but two instances are solved.

Table 3 shows the importance of the ordering heuristics. The first columns show the full model using the old ordering heuristics, i.e. first-fail and short-connections-first, while the last columns show the full model with the route-fixing ordering. When using the old heuristics, we have applied the pricing constraint propagation every #flights/#aircraft search nodes, to approximately apply is once for every fixed aircraft. This was to get a fair comparison with the route-

Table 2. Comparison of the basic model with and without reachability propagation.

	Basic model		With reach.	
Instance	BTs	Time(s)	BTs	Time(s)
Instance 1	2965*	122	18	63
Instance 2	2388*	73	2	91
Instance 3	0	57	1	74
Instance 4	2757*	74	2379*	450
Instance 5	2379*	74	44	75
Instance 6	2839*	76	29	71
Instance 7	2474*	786	5	58
Instance 8	2604*	72	34	67
Instance 9	2858*	467	2604*	1283
Instance 10	2040*	63	3	54
Instance 11	2232*	102	17	68
Instance 12	26	70	2	68
Instance 13	2163*	69	591	67

fixing odering, which applies the propagation exactly once per route. It is clear from the table that the old ordering heuristics do not work well at all, even in the presence of reachability propagation and the pricing constraint. No solution is found, and for all problems the excessive thrashing gives rise to very long running times.

8 Conclusions

We have presented a full constraint model for the Tail Assignment problem. The model includes a reachability algorithm that provides efficient propagation required to capture the flight restriction rules. The model also uses the pricing module from a column generation approach to Tail Assignment [10] to model maintenance rules, which would otherwise require substantial effort to re-implement. It has previously been shown how constraint programming techniques can improve the performance of column generation for this problem [11], and this model, by re-using a column generation pricing problem to check feasibility and improve ordering heuristics, shows that the reverse is also possible. Further integration between the approaches will likely benefit a lot from the results presented here, and will be researched further. We also conjecture that adding future constraints to a Tail Assignment model will often be easier for the constraint model than for the full column generation model, making it an excellent tool for experimentation.

One potential drawback of the maintenance rule handling is that the propagation is not that strong. However, since these constraints are seldom extremely tight in practice, the limited propagation is enough to achieve reasonable performance. And the fact that the handling of these rules has not been re-implemented

Table 3. Comparison of constraint models using old ordering heuristics, no reachability propagation, and the full model.

	Full model, old ordering		Full model, new ordering	
Instance	BTs	Time(s)	BTs	Time(s)
Instance 1	2388*	1182	177	70
Instance 2	1391*	1878	0	98
Instance 3	2757*	1238	0	82
Instance 4	2379*	1353	146	99
Instance 5	2839*	1191	116	78
Instance 6	2652*	1755	0	99
Instance 7	2474*	1458	0	68
Instance 8	1268*	1867	0	106
Instance 9	2604*	1227	0	66
Instance 10	2040*	1071	0	112
Instance 11	2226*	1172	0	112
Instance 12	2163*	752	0	134
Instance 13	2232*	1009	0	100

in the constraint model is immensely important, since the described implementation is part of a production system.

It is possible that the reachability algorithm, as well as the idea of re-using a column generartion pricing problem, could be of use also for other types of applications, similar to Tail Assignment. Finally, it should be mentioned again that this model is not designed to work alone to solve the Tail Assignment problem. It is primarily designed to quickly produce a feasible initial solution that can then be improved, given a cost function, using a combination of column generation and constraint propagation.

References

[1] L. W. Clarke, E. L. Johnson, G. L. Nemhauser, and Z. Zhu. The Aircraft Rotation Problem. *Annals of Operations Research*, 69:33–46, 1997.

[2] B. de Backer, V. Furnon, P. Kilby, P. Prosser, and P. Shaw. Solving Vehicle Routing Problems using Constraint Programming and Metaheuristics. *Journal of Heuristics*, 1(16), 1997.

[3] G. Desaulniers, J. Desrosiers, Y. Dumas, M. M. Solomon, and F. Soumis. Daily Aircraft Routing and Scheduling. *Management Science*, 43(6):841–855, July 1997.

[4] M. Desrochers and F. Soumis. A generalized permanent labelling algorithm for the shortest path problem with time windows. *INFOR*, 26(3):191–212, 1988.

[5] M. Elf, M. Jünger, and V. Kaibel. Rotation Planning for the Continental Service of a European Airline. In W. Jager and H.-J. Krebs, editors, *Mathematics – Key Technologies for the Future. Joint Projects between Universities and Industry*, pages 675–689. Springer Verlag, 2003.

[6] G. Erling and D. Rosin. Tail Assignment with Maintenance Restrictions - A Constraint Programming Approach. Master's thesis, Chalmers University of Technology, Gothenburg, Sweden, 2002.

[7] T. Fahle, U. Junker, S. Karisch, N. Kohl, M. Sellmann, and B. Vaaben. Constraint Programming Based Column Generation for Crew Assignment. *Journal of Heuristics*, 8(1):59–81, 2002.

[8] M. Gamache, F. Soumis, G. Marquis, and J. Desrosiers. A Column Generation Approach for Large-Scale Aircrew Rostering Problems. *Operations Research*, 47(2):247–263, April-March 1999.

[9] R. Gopalan and K. T. Talluri. The Aircraft Maintenance Routing Problem. *Operations Research*, 46(2):260–271, March-April 1998.

[10] M. Grönkvist. Tail Assignment — A Combined Column Generation and Constraint Programming Approach. Lic. Thesis, Chalmers University of Technology, Gothenburg, Sweden, 2003.

[11] M. Grönkvist. Using Constraint Propagation to Accelerate Column Generation in Aircraft Scheduling. In *Proceedings of CPAIOR'03*, May 2003.

[12] C. Halatsis, P. Stamatopoulos, I. Karali, T. Bitsikas, G. Fessakis, A. Schizas, S. Sfakianakis, C. Fouskakis, T. Koukoumpetsos, and D. Papageorgiou. Crew Scheduling Based on Constraint Programming: The PARACHUTE Experience. In *In Proceedings of the 3rd Hellenic-European Conference on Mathematics and Informatics HERMIS '96*, pages 424–431, 1996.

[13] C. Hjorring and J. Hansen. Column generation with a rule modelling language for airline crew pairing. In *Proceedings of the 34th Annual Conference of the Operational Research Society of New Zealand*, pages 133–142, Hamilton, New Zealand, December 1999.

[14] E. Kilborn. Aircraft Scheduling and Operation — a Constraint Programming Approach. Master's thesis, Chalmers University of Technology, Gothenburg, Sweden, 2000.

[15] J.-C. Régin. A filtering algorithm for constraints of difference in CSPs. In *Proceedings of AAAI-94*, pages 362–367, 1994.

[16] L.-M. Rousseau, M. Gendreau, and G. Pesant. Using Constraint-Based Operators to Solve the Vehicle Routing Problem with Time Windows. *Journal of Heuristics*, 8(1):43–58, 2002.