Aircraft Assignment: A Basic CP Model

Erik Kilborn
Computing Science, Chalmers University of Technology
SE-412 96 Göteborg, Sweden
ek@cs.chalmers.se
http://www.cs.chalmers.se/~ek

Abstract. In this paper we model the aircraft assignment problem using constraint programming. Route planning problems are often modelled using column generation. By using CP, the problem can be formulated in one single model that fully exploit the network structure of the problem. We look at a simplified formulation of the problem, how to model it, using efficient propagation, and how to build a fast construction heuristic from it.

The Problem

Once the timetable of flights has been constructed, an airline has to plan its flying resources: aircraft and crew. Here we will look at aircraft scheduling. The airlines solve the problem in a two step procedure. The first step, *fleet assignment,* assigns an aircraft type (e.g. Boeing 767) to each flight. Once this step is finished, the problem decomposes into one subproblem per fleet. The remaining problem is to decide which individual aircraft should operate each flight, and at the same time specify the rosters for the aircraft. This is known as the *aircraft assignment* problem, or the *tail assignment* problem.

The aircraft assignment problem can be characterized as:

• *input*: a set of aircraft + a set of flights + a set of pre-assigned maintenance checks

• *output*: assign all flights to aircraft, creating rosters for the aircraft respecting connection constraints and maintenance constraints.

There are two types of activities: flights and maintenance checks. An activity is defined by a tuple `(start_time, end_time, start_airport, end_airport)`. For example `(15:35 02APR2001, 17:05 02APR2001, FRA, LHR)` defines a flight departing 15:35 from Frankfurt, arriving 17:05 in London Heathrow, April 2, 2001. For maintenance checks `start_airport` and `end_airport` will always be the same.

A *roster* is the whole sequence of activities assigned to the same resource. An aircraft roster is also a route, where the aircraft travels from airport to airport.

A *pre-assignment* is an activity which is fixed to a specific resource in the input to the problem.

The *connection constraints* defines when two activities can be placed directly in sequence in a roster. Between any two activities *i* and *j* there is a required minimum connection time $\texttt{min\_connection\_time}_{ij}$. Activity *i* can be directly followed by activity *j* in a roster, if:

1. $\texttt{end\_airport}_i = \texttt{start\_airport}_j$, and
2. $\texttt{end\_time}_i + \texttt{min\_connection\_time}_{ij} \leq \texttt{start\_time}_j$

The length of the minimum connection time depends on many attributes, mainly on the airport and the aircraft type. Typically the required turnaround time for an aircraft is between 30-60 minutes.

Apart from the restrictions in time and space, the rosters of the aircraft have to follow *maintenance constraints*, which specify that an aircraft must not fly too long without getting maintenance. There are different kinds of maintenance checks, spanning from the ones that takes a couple of hours—normally over night—and should take place e.g. every 150th flight hours, up to the checks that take several weeks and occur only once or twice in a decade. Maintenance checks fall into two categories: minor checks and major checks. The minor checks have to be

placed in the rosters, when needed, while solving the problem. The major checks are planned longer in advance, and there is no room for moving them when solving the aircraft assignment problem, which is typically not solved for periods longer than a month. Therefore the major checks are modelled as pre-assignments.

In this paper, we will leave out the minor checks, and only include the maintenance checks that can be modelled as pre-assignments (Under certain circumstances, e.g. if the scheduling period is just one day, also the minor checks can be modelled as pre-assignments).

Modelling

An elegant way to model all the connection constraints, is to build a *connection network*. A connection network is a time-directed graph with the nodes representing the activities, and the arcs representing the possible connections between activities, according to the connection constraints. Since all activities are fixed in time, there will be no cycles in the graph. Once the connection network has been built, we need not remember the formulation of the connection constraints any more. A roster is a path in the graph. Since each roster must belong to a specific resource, special start nodes for each resource has to be introduced in the graph. We let these start nodes correspond to the *carry-in* activities, i.e. the last activity in the previous scheduling period for each resource. The carry-in activities are the starting points for each aircraft and constrains which can be the first activity assigned in the roster. There are several options of how to model the end nodes in the graph. We choose to let the end nodes be the same as the start nodes, so that the end of a roster is modelled by linking back to its start node.

Each activity should be assigned to exactly one resource. In terms of the connection network, that means that each node should have exactly one incoming and one outgoing arc, i.e. each activity should have exactly one predecessor and one successor in a roster. A solution to the aircraft assignment problem is a set of paths in the connection network such that each node is visited exactly once, where the number of paths equals the number of available resources. Finding such an assignment is a network flow problem, and can be solved in polynomial time. If it had not been for the maintenance regulations, the whole aircraft assignment problem would have been this simple. This network flow problem is an important *core problem* of aircraft assignment.

We transform the connection network $G=(V,A)$ to a bipartite graph $B=(V_{out},V_{in},A)$, where V_{out} is a duplicate of node set V keeping only the outgoing arcs of the nodes, and V_{in} is a duplicate of node set V keeping only the incoming arcs. The core problem can now be formulated as a *bipartite matching* problem. A feasible solution to the aircraft assignment problem must be a perfect matching in the bipartite graph. In $O(|V|^{1/2}|A|)$ such a matching is found, or we can establish that no perfect matching exists [3]. If the arcs are given weights to reflect the preference of the connection, an optimal solution to the core problem can be obtained in $O(|V|^3)$ [3]. This is known as the simple assignment problem.

In our CP approach we express the network structure of the problem by introducing decision variables denoting the successor of each activity. We refer to them as *next*-variables, and their domains are setup according to the connection network. For each activity, the corresponding node is identified in the network, and the domain of the *next*-variable representing this activity is setup to be all the activities (nodes) to which it has outgoing edges. Each activity is represented by a unique integer number. We collect the *next*-variables in an array. The *next*-array represents the same information as the bipartite graph *B*, which represents the same information as the connection network *G*. The core problem is now expressed by applying an *alldifferent*-constraint over all the *next*-variables. The *alldifferent*-constraint states that a solution to the problem must be a perfect matching in the bipartite graph—or in terms of the connection network, that each activity must belong to exactly one path, i.e. to the roster of exactly one aircraft.

Now we need to model pre-assignment constraints of the form "Activity A_{pre} is fixed to resource R_i". We have to make sure that the path we get, by following the *next*-variables from the start node of R_i, passes through activity A_{pre}. This can be modelled as a simple kind of path constraint: *an activity and its successor must both be allocated to the same resource*. In order to express this we introduce a new set of variables, one for each node. We will refer to them as *alloc*-variables. Their domains represent the set of aircraft the activity may be assigned to. We fix the *alloc*-variables of the pre-assignments to the aircraft defined by the pre-assignment constraints. We fix the *alloc*-variables of the carry-in activities to their corresponding aircraft. The relation between the *next*- and the *alloc*-variables can now be expressed as

alloc[i] = alloc[next[i]], for all nodes *i* in our connection network.

Thereby we have taken care of the pre-assignment constraints in the model.

Let us summarize the model. Let *n* be the number of activities (flights + pre-assignments), and *m* be the number of aircraft. Build a connection network of *m* start nodes and *n* activity nodes. Let the end nodes be the same as the start nodes, meaning that each roster will become a cycle in the graph. In the model, we will let the indices of the start/end nodes be the same as the indices for the corresponding resource respectively. Then the model is:

Objects:

m resource nodes (start/end nodes), enumerated as *1,...,m*, and
n activity nodes, enumerated as *m+1,...,m+n*

Variables:

An array of *next*-variables: *next[1...m+n]*,
with each domain setup to the set of successors of the corresponding node in the connection network.

An array of *alloc*-variables: *alloc[1...m+n]*, with domains initiated to:
for nodes *i=1,...,m*, *dom(alloc[i]) := {i}*[1],
for nodes *for i=m+1,...,m+n*, if activity *i* is pre-assigned to resource *r*
then *dom(alloc[i]) := {r}*, else *dom(alloc[i]) := {1,...,m}*

Constraints:

alloc[i] = alloc[next[i]], for *i=1,...,m+n*
alldifferent(next)

We have expressed the whole problem in one model, which can be extended with additional maintenance regulations. With an OR approach we would have difficulties in doing so, since the relations do not let themselves be expressed as linear constraints. Typically, in an OR approach, when it is hard or impossible to express a problem in one single model, we end up using column generation. This method has it merits, but it also means that we lose the global view of the problem. In this case, we would not be able to exploit the fact that bipartite matching is a dominating subproblem. Still, for the CP model to be competitive we need to find: i) efficient ways of propagating the constraints in the model, and ii) good lower bounds for the cost function. In this paper we will focus on the propagation, leading us to a construction heuristic for the problem.

Propagation

The *alldifferent*-constraint expresses the core subproblem of aircraft assignment. Régin's filtering algorithm provides a complete and efficient propagation algorithm for the *alldifferent*-constraint. It will serve as the main motor in our solution method.

1. The notation *dom(x)* is used to say "the domain of variable *x*".

Internally among the *alloc*-variables, propagation is based upon that overlapping activities cannot be allocated to the same resource.

How to deal with the *alloc[i] = alloc[next[i]]* constraints? Each such constraint involves several variables: first we have *alloc[i]* and *next[i]*, and in addition to that all *alloc[j]* variables where $j \in dom(next[i])$. The constraint tells us to respect the following rules:

1. $dom(alloc[i]) \subseteq \bigcup_{j \in dom(next[i])} dom(alloc[j])$

2. $\forall j \in dom(next[i]),\ dom(alloc[i]) \cap dom(alloc[j]) \neq \emptyset$

If the first rule is violated, consistency is restored by domain reduction of *alloc[i]*. For the second rule, if the intersection becomes empty for a certain *j*, consistency is restored by removing the value *j* from the domain of *next[i]*.

An attempt was made to use propagation that triggers for any domain change of any of the involved variable, but it showed to be inefficient. The time spent to administer the propagation did not pay off. The many variables involved in the constraint made it trigger often, but the majority of the times no domain reductions could be done. Instead, propagation is not started until either *alloc[i]* or *next[i]* is fixed to a value:

- When the value of *next[i]* is fixed (to *succ_of_i*) we get an ordinary equality constraint, of the form *alloc[i] = alloc[succ_of_i]*, (only two variables involved).
- When the value of *alloc[i]* is fixed (to *resource_of_i*) we get a constraint of the type *resource_of_i = alloc[next[i]]*

  The initial propagation procedure for the latter is as follows:

```
foreach j ∈ dom(next[i])
       if resource_of_i ∉ dom(alloc[j])
       then remove j from dom(next[i]);
```

After that, we need to watch each *alloc[j]* for removal of *resource_of_i*, which must then lead to removal of *j* from *dom(next[i])*.

By dynamically adding these simplified versions of the original constraint when a variable has been fixed, an efficient trade-off is found for the propagation of the constraint, and thereby for the model as a whole.

Application

We are now ready to build a CP solver for aircraft assignment, based on our model. A solver was implemented using ILOG Solver. The work was performed at Carmen Systems AB, which is a software developer of optimization tools for the airline and rail industry. Test problems from a mid-size North American airline were available for testing—the biggest one for a short-haul fleet of 17 aircraft, a scheduling period of one month, 3031 flights to assign and 12 pre-assigned maintenance checks. In all of the test problems, the percentage of pre-assigned activities in the input were less than 1%. This gives a hint about how dominating the network flow structure is in the problem.

In all practical cases, inconsistencies in the input data was found simply by the initial propagation of the *alldifferent*-constraint. Such inconsistencies have two main sources: 1) unbalanced airports, for which the number of incoming and outgoing flights are not the same in the scheduling period, and 2) misplaced pre-assignments, so that there is not enough capacity on a specific day. Both these types of inconsistencies are detected by the *alldifferent*-constraint in a few seconds, even for the biggest problems. This is an important property of the CP solver since in a practical situation the user of a planning system will spend a lot of time getting the input

data set up correctly. In order to help the users, the model was extended with the ability to cancel flights, so that the solver returns a solution for any input. Before, the solver would simply say "not feasible" for inconsistent inputs, and refuse to do anything more. Now it will instead return a solution accompanied with a list of unassigned flights. By studying the list of unassigned flights, the user will easily identify an unbalanced airport or a misplaced pre-assignment. Cancellations are modelled by extending the domains of the *next*-variables, letting *next[i] = i* mean that flight *i* is cancelled. In terms of the connection network, this means adding a self-loop for each node. Also the *alloc*-variables and the *alloc[i] = alloc[next[i]]* constraints require extensions.

Since the network structure of the problem is dominating and has the most effective propagation, the *next*-variables are chosen as *decision variables*, i.e. the ones to instantiate in the search. The other possible choice are the *alloc*-variables. Notice that an instantiation of the *alloc*-variables also uniquely defines a solution.

We have not defined a *cost function* yet. Clearly, the most important aspect costwise is that all flights get assigned. Our solver will always achieve this when possible. But there will be many feasible solutions. Which ones do we prefer? In terms of connections, the worst ones are the ones of medium length (around 2-4 hours of connection time). Short connections are good since there is little waste time, but actually also long connections are good since the aircraft can serve as stand-by in these gaps. We should realize that there is no use in trying to minimize the total connection time in a solution. The problem is a type of two-dimensional bin-packing and the amount of space (connection time) in a solution will always be the same. From this we also learn that the more short connections we have, the fewer are the non-short connections (and the longer they are). If we greedily look for short connections, we increase the probability of finding a feasible solution in that search branch. Just like in a greedy heuristic for TSP we will also be left with very long connections by the end of the search. But here the long connections are actually good!

The *value ordering* of the search is therefore defined as: The shorter the connection time, the higher the priority. If there are no more activities in the domain, the value indicating end of roster is tried, and as a last resort cancellation of the activity. The *variable ordering* basically follows the principle of smallest domain first, but pre-assignments are given higher priority, since they cannot be cancelled.

Based on this we can build a simple but fast construction heuristic for the problem. The search procedure is traditional depth-first search with chronological backtracking. As soon as we find a solution to the problem, we let the search stop. If we keep all the cancellation possibilities in the model, the propagation does not become effective. It will be a feasible solution to cancel all flights, or most of the flights, so the *alldifferent*-constraint will not be able to exclude many connections. Therefore, we let the solver first search without allowing cancellations. If no solution is found, we have proven that there is no solution with all activities assigned. Then the search is restarted with the cancellation possibilities included.

Results[1]

The test were run on a 440 MHz HP workstation. For the biggest test problem with 17 aircraft and over 3000 activities it takes the CP solver 20 seconds to find a solution. For an inconsistent problem of the same size, where cancellations are necessary, a solution is provided in around a minute [2]. A reason for the solver being fast is that it rarely backtracks, we might say that it behaves in a quasi-greedy manner. Empirical observations, based on the test problems, show the following:

1. More numerical results will be prepared for the final version of this paper.

- The amount of backtracking nodes in the search tree if on average 0.3% of the number of variables to instantiate. For the biggest problem the solver backtracks only 5 times.
- On average only 30% of the variables has to be actively instantiated, the remaining 70% became fixated by propagation.

The complexity of Regin's algorithm along one search branch is $O(|A|^2)$, where A is the set arcs in the connection network. Or differently expressed: $O(n^2d^2)$, where n is the number of activities (*next*-variables), and d is the size of the largest occurring domain [4]. In practice, our solver has not performed worse than this [2]. Recall that the amount of backtracking is minimal, so the solution is virtually found by just following one search branch. Apparently, the trade-offs for the other propagation procedures in the solver are well balanced. These observations are based on a limited set of test data from just one airline, so they should be considered as very preliminary. Also, this is only valid for real-world problems. A reason for good performance on real-world data is obviously that the timetable of flights, that is the input to the problem, intentionally has been constructed in a way to make it easy to build aircraft routes from it. It is easy to intentionally construct a problem so that the solver performs badly. An interesting observation is that for a fixed number of aircraft, the runtime in practice grows with n^2 for an input of n activities, when we increase the size of the problem by extending the scheduling period [2].

Work in progress

- More empirical data from more airlines, and more analysis of the results.
- Completion of the model. Path constraints for minor maintenance checks. Extension and refinement of the cost function, including e.g. crew considerations.
- Building an optimization method on the model, by wrapping up the search in branch-and-bound. Lower bounds are provided by solving the simple assignment problem.
- Comparing the CP approach with the column generation approach developed at Carmen Systems, and possibly combining the two, e.g. by using the CP construction heuristic to create a good set of initial columns for the column generator.
- Construction of a method for repair of aircraft rosters to be used on the day of operation. Here we want solutions close to the original solution. Also we want several options for solutions, since a solution might not be feasible for other resource types (crew, passengers, etc.). A breadth-first local search starting from the original solution is used (limited discrepancy search layer by layer).
- The major challenge is to extend the model with time variables, and to find efficient propagation for this. Time variables are necessary to being able to retime flights when repairing schedules at day of operation, but can also be of use when solving tight aircraft assignment problems.

References

1. M. Grönkvist, *Aircraft Scheduling*, Internal Report, Chalmers University of Technology, 2000

2. E. Kilborn, *Aircraft Scheduling and Operation: a Constraint Programming Approach*, Master's Thesis, Chalmers University of Technology, 2000.

3. C.H. Papadimitriou, K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Dover Publications 1998.

4. J-C Régin. A filtering algorithm for constraints of difference in CSPs. In *Proceedings of AAAI-94*, Seattle, Washington, 1994, pp 362-367, 1994.