Report on the implementation of the Aircraft Landing Problem

Modeling and Solving Constrained Optimization Problems

Bernhard Mallinger
0707663

January 19, 2013

1 The problem

This exercise deals with implementing the Aircraft Landing Problem, which is specified by the following constraints:

TIME Each aircraft has a minimum, preferred and maximum time of landing.

COST There are possibly different costs for airplanes being too late or too early.

RUNWAY Runways can be unavailable for certain periods.

SEQUENCE DELAY Sequences of airplane landings have to keep a certain offset depending on the airplane types.

TYPE SEQUENCE There must not be more than 3 landings of the same airplane type in sequence and in any sequence of 5 landings, there must be at most 2 medium sized and 1 large airplane.

LANDINGS IN 30 MIN There is a limit to the number of landings that can take place in 30 minutes.

2 Basic constraints

To get started, I decided to select a subset of basic constraints for implementation and testing with different parameters in order to get acquainted with constraint programming and Gecode and to get a feel for which decisions lead to correct results and furthermore which of the ones that are correct are actually computationally feasible.

TIME The first decision was to represent the times when aircrafts land as array of concrete periods `aircraftTimes` of the length of the number of aircrafts. Each entry corresponds to one aircraft. This gives a natural way of describing the time frame, in which it is possible for them to land, by specifying the domain for each entry to be this time as given by the instance.

COST This representation also allows to model the cost function quite easily. Since being late and early is punished by different costs, it is necessary to calculate their values separately. The amount of for instance delay for an aircraft `i` is given by `max(0, aircraftTimes[i] - instance.aircraft[i].preferredTime)`. Using this, an array of delays can be constructed, which is suitable for use with the `linear`-constraint in combination with the cost vector. The resulting value summed up with an analogously constructed value for being too early make up the cost value.

LANDINGS IN 30 MIN In order to represent the number of landings within a certain time frame, a different representation has to be found. As this constraint concerns ranges of periods rather than aircrafts, I employed a set array `timeAircrafts`, whose entries represent the set of airplanes landing at the respective periods. It is linked to `aircraftTimes` by requiring that the index of the airplane `i` is an element of `timeAircrafts[ aircraftTimes[ i ] ]`. After formulating it on a low level using `element` and `rel` constraints, I discovered that there is a `channel`-constraint which implements exactly this behaviour andt results, depending on the instance, in faster runtimes of about a third. Now it's only necessary to sum up the cardinalities of these sets for each time frame of 30 minutes and stating that this sum has to be smaller than the number given by the instance.

Since this simple constraint, if it is tight enough to actually matter, already leads to very bad runtimes for small instances (20-30 planes), I decided to try to vary it in order to allow for better propagation. In the variation, the union of all sets of airplanes for each 30-minute-frame is calculated in a set variable which whose maximal cardinality is bounded by the limit given by the instance. As this change does not change the semantics, the number of solutions and failed nodes stays the same, but depending on the instance, the runtime is sometimes noticeably affected with some instances being solved nearly twice as fast whereas others show no significant change. This suggests that the internal propagation mechanism of Gecode can in some cases deal more efficiently with the second variation.

In order to calculate the union of all sets of airplanes of the corresponding time frame,

I tried two different formulations: One by requiring that each of the relevant sets to be a subset of the union, and the other by using the union of the suitable `slice` of the array. Interestingly, the variant using the `slice`-operation, even by being on a somewhat higher level and therefore potentially giving Gecode more possibilities for optimisation, is slower by a factor of about 2.

3 Investigations on branching

Based on these constraints, it is possible to do investigations on the branching parameters. For branching, strategies can be selected for which variable to branch on as well as which values of the domain to pick. It is easily empirically evident that these parameters change the required computation time by multiple orders of magnitude and decide whether a formulation finishes within instant or is computationally infeasible.

Due to the size of the search space, comprehensive measurements and tests with larger instances or selectors, that seem less promising, would exceed the scope of this assignment. Therefore a focus has been fixed on on manageable instance sizes and certain cases that exhibit interesting behaviour.

Instances with no severe constraint are usually solved quickly using the `INT_VAR_NONE`-selector, which selects unassigned variables first. However on instances, that are already tightly constrained by the **LANDINGS IN 30 MIN**-constraint, the `INT_VAR_AFC_MAX`- along with the `INT_VAR_SIZE_MIN`-selector has shown to be exceptionally efficient. They have in common that they provoke quick failures by chosing variables which have often lead to failure already or which have minimal domain size respectively, thereby cutting the search space quickly. Other selectors preferring variables with large domains lead to disastrous runtimes up to the point where it makes no sense to include them in this investigation.

8 aircrafts, 3 per 30 min	`VAR_AFC_MAX`		`VAR_SIZE_MIN`		`VAR_NONE`	
	time	nodes	time	nodes	time	nodes
`VAL_MAX`	0.78	3054	0.53	2141	6.80	51245
`VAL_MED`	0.32	1807	0.18	563	9.74	74163
`VAL_SPLIT_MAX`	1.75	12361	0.57	2376	7.31	59378

Table 1: Measurements of the run time behaviour on an instance containing 8 airplanes scheduled in 120 periods with a maximum of 3 per 30 minutes. The time is given in seconds, "node" means the number of nodes expanded during the search.

Table 1 shows measurements using different combination of promising selectors. As not many constraints have been posted so far, ordering based on the degree of variables somewhat

corresponds to having no special variable selection order. Therefore other selectors not mentioned here, which are geared towards quick failures and are based on the node degree, do not have significant effects in this case.

Since preferred values for landing times of common instances are somewhere between the minima and maxima, choosing values around the extremes of the domain is not optimal as the direct comparison of `INT_VAL_MAX` and `INT_VAL_MED` shows, where the latter is faster by a factor of roughly 2 for the first two selectors. Choosing values that split the domain are not particularly efficient as the results show for `INT_VAL_SPLIT_MAX` (`INT_VAL_SPLIT_MIN` showed a comparable behaviour and is therefore omitted in the table)

Due to these investigations, future tests are mostly conducted using `INT_VAR_SIZE_MIN` or `INT_VAR_AFC_MAX` in combination with `INT_VAL_MED` since they have proven their efficiency in some instances of the problem, but for other cases, other configurations might very well be superior. In fact on some larger instances, `INT_VAR_AFC_MAX` outperforms `INT_VAR_SIZE_MIN` by an order of magnitude, which suggests that in these cases, variables with small domains are not necessarily the ones that often lead to quick failures, but rather the accumulated failure count statistics is more capable of determining the promising ones. This observation however cannot be generalized as there are counterexamples, in which `INT_VAR_AFC_MAX` yields problems at a certain stage during the optimisation. There, it is reasonable to assume that certain choices in the current partial assignment lead to a different situation, which invalidates the empircal data collected in other branches with different partial assignments. Since this old data is still used for decisions, they are not optimal with respect to the current situation.

In order to profit from the strengths of these two settings, `tiebreak(INT_VAR_SIZE_MIN, INT_VAR_AFC_MAX)` can be utilized. There are instances where it outperforms each setting individually by far, however this is again does not hold in general. The effects of the different settings have been observed to very much depend on the input data.

4 Further constraints

RUNWAY

Up to this point, the assignment to runways has not been considered. To model the availability of runways, it is necessary to not only assign the planes to runways, but also to combine it with temporal information, i.e. airplanes only cannot be assigned to runways during certain periods of time. As first step, aircrafts are assigned to runways using a vector `aircraftRunways` with a domain element for each runway similar to `aircraftTimes` for time slots.

Then, information about time is taken into account in a (conceptually) two-dimensional array `timeAircraftsRunways`, whose dimensions are time ($0 \ldots$ number of periods) and runways ($0 \ldots$ number of runways). A simple `element`-constraint simulating the two dimensions

forces correspond entries of aircrafts to their index `i`: `element(timeAircraftsRunways, (periods * aircraftRunways[i]) + aircraftTimes[i], i)`. As a side-effect, it is now ensured that no two planes can land on the same runway at the same time. Now it is conveniently possible to model runway unavailabilities by setting the respective values for an aircraft to an invalid value such as `-1`.

SEQUENCE DELAY

Since the required delays between aircrafts depend not only on their types, but also on the sequence of their types, it is necessary to encode information about the sequence in the model, whereas the type information is statically known. This constrained can be expressed by an implication: For each pair of airplane `i` and `j`, `i`≠`j`, if `i` lands before `j` on the same runway, the landing time of `i` has to be smaller then the landing time of `j` minus the delay depending on the types of `i` and `j`. Since implications are essentially disjunctions, this makes it harder to consider branches as failed and in fact takes many instances of more than about 15 airplanes to the boundary of feasibility.

It is possible to optimise these constraints by using logical simplifications in order to combine constraints: If the runways differ, neither direction of the delay needs to be considered. Furthermore, since the airplanes cannot land at the same time as ensured by **RUNWAY**, we can deduce that if one airplane lands before the other one, the other one does not land before the first one and vice verca. Therefore the constraint can be reduced to `!sameRunway || ( (isBefore_ij && maintainsDelay_ij) || maintainsDelay_ji )` and only needs to be stated for each `i` and `j` with `i<j`, thereby halving the number of constraints. The effect of this optimisation strongly depends on the input instance: There are some where the speedup is negligible, on one particular instance a reduction of the number of expanded nodes from 5.5 million to just 7 thousand has been observed (resulting in a wall clock time reduction from 7 minutes to one second). The general picture of tests with these variants suggest that the impact of this constraint on the search space is a measure of the amount of effect of this optimisation.

This formulation however still is hardly feasible for instances with more than 20 aircrafts. As another variation, I tried to instead of using time aircraft landing time of the `aircraftTimes`-array to use the position in `timeAircraftsRunways`. This way, it would be possible to skip the check for runways, since the index of aircrafts landing in different runways is far apart and therefore automatically satisfies one one the delay requirements (except for border cases at the beginning and end of the time frame). However getting rid of this one disjunct this proved to be clearly less efficient as it requires extra work in calculating the index in `timeAircraftsRunways` (even with the relaxation of not considering the mentioned border cases).

As further optimisation, it is possible to omit constraints for those pairs of aircrafts, whose minimum and maximum landing times are that far apart that it there is no valid assignment which violates the **SEQUENCE DELAY**-constraint. This however only has negligible effect.

TYPE SEQUENCE

The final constraint to be considered is **TYPE SEQUENCE**. Gecode offers a fairly intuitive way of representing this type of constraints, which is `sequence`. It restricts the number of occurrences of elements of a given set in sequences of a given length. So to disallow sequences of more than 3 aircrafts of the same type, it can be specified that in each sequence of length 4, at most 3 are allowed to be of the set of aircraft ids of an homogeneous type. Constraints for medium and large aircrafts can be directly expressed by allowing only 1 or 2 out of any sequence of 5 aircrafts to be of a set of aircraft ids of the respective type.

The difficult part of this aspect however is to obtain an array `aircraftSequence` in which the aircrafts are represented by their id (such that their type can be inferred) and ordered by their landing time. There does not seem to be a designated channeling variant for this purpose, so I created an array with distinct integers up to the number of aircrafts, for which the condition holds that for all adjacent elements, the value in `aircraftTimes` when using the element as index yields a smaller or equal value for the left element when compared to the respective value of the right element. This ensures that in `aircraftSequence`, the aircrafts are actually ordered by their landing time. To actually find the values of this array, it is however necessary to branch on it. This happens only as final step of the computation (i.e. the branching statement is last), as otherwise, it would impose an ordering uninformed of the landing time and other constraints, which also empirically has been observed to be hardly feasible.

This constraint eliminates a large number of solutions of instances of the instance generator, which generates large aircrafts with a probability of $\frac{1}{3}$. For this reason, I altered the test data to contain more smaller aircrafts in order to actually being able to find a solution. However even with this simplification, instances comprising more than about 10 aircrafts have shown to be very hard to solve in reasonable time on a conventional machine.

5 A different approach to finding the optimum

So far, the optimal solution has been calculated by using Gecode's branch and bound feature. It utilises bounds from valid solutions that it generates and posts them as upper limit for the cost when continuing the search process. Since the search is not guided, it happens that a number of solutions are found that are far away from the optimum, which contain solution fragments with a high cost. With this in mind, the original idea can be reversed and instead of starting with no cost bound at all, the cost can be constrained to a very small number. This way, only a very narrow solution space is explored where expensive solution fragments are quickly disregarded (i.e. only a few airplanes are allowed to deviate from their preferred landing time). If the instance is suitable for this search process, it can be very effective. This is however not the case for instances with a costly optimal solution (except if it is possible to estimate a tight upper bound for said value).