Algorithms

Francis

February 14, 2021

The Constructive Heurist for the Aircraft Recovery Problem (CHARP) is based in Constraint Satisfaction Programming (CSP) and consists of two heuristics, that perform a pincer movement over the lower and upper bounds of the search space. Given the original aircraft rotation and a set of disruptions, the objective of the CHARP is to create a new combination of aircraft routings during the RTW to minimize the total cost of recovery.

The algorithm starts by initializing all the data sets after which it will loop through each aircraft and their rotations. After initializing the rotation the algorithm checks the rotation for cancelled flights or aircraft breakdown periods that also lead to flight cancellation. If these disruptions exist the algorithm will then create new flights.

Algorithm 1 describes the creation of new flights. The algorithm receives as inputs the aircraft rotation, the distance set, the maximum flight number, the aircraft breakdown period and the end time of the recovery window. The algorithm will return the aircraft rotation with the new flight and the updated maximum flight number. This algorithm starts by initializing two sub-rotations containing available flight slots, the cancelled flights and the starting time from which new flights can be created. The algorithm will afterwards loop through the new flight slots and cancelled flights and calculate the departure time and the flight time (line 6). If the departure time added to the flight time of the new flight over shoots the end time of the recovery time window the loop breaks and the algorithm returns the rotation with the new flights and the number of the last flight. Else, the flight number is incremented a and the new flight slot is updated (lines 11 to 15). Finally the algorithm checks if the new flight's arrival time added with the transit time overshoots the end time of the recovery time window. If true the algorithm breaks the loop and returns the rotation with the new flights and the number of the last flight. An identical algorithm can be derived from 1 by simply allocating the new flight slots to those that were cancelled due to flight disruption.

Algorithm 1: New flights from aircraft disruption

```
Input: $\sigma_p, \Delta, M, \mathcal{B}_b, RTW_e$
Output: $\sigma_p, M$
1  $\sigma_p^n \leftarrow$ List of available new flights in $\sigma_p$
2  $\sigma_p^c \leftarrow$ List of cancelled flights in $\sigma_p$
3  $start \leftarrow$ end time of aircraft disruption $\mathcal{B}_b$
4  for $\sigma_p^n(i), \sigma_p^c(i)$ in $\sigma_p^n, \sigma_p^c$ do
5      if $i \neq 0$ then
6          $start \leftarrow \sigma_p^{na}(i-1) + t_{rp}$
7      $\delta_{of} \leftarrow \Delta(\sigma_p^{co}(i), \sigma_p^{cf}(i))$
8      if $start + \delta_{of} > RTW_e$ then
9          break
10     $M \leftarrow M + 1$
11     $\sigma_p^n(i) = M$
12     $\sigma_p^{nd}(i) \leftarrow start$
13     $\sigma_p^{no}(i) \leftarrow \sigma_p^{co}(i)$
14     $\sigma_p^{na}(i) \leftarrow start + \delta_{of}$
15     $\sigma_p^{nf}(i) \leftarrow \sigma_p^{cf}(i)$
16     if $\sigma_p^{na}(i) + t_{rp} > RTW_e$ then
17         break
18 return $\sigma_p, M$
```

The next step uses algorithm 2 consists in traversing the rotation to find if the following five constraints are being respected:

- Flight schedule continuity.
- Transit or turnround time between consecutive flights.
- Departure and arrival airport capacity.
- Aircraft arrive on time for maintenance.

The algorithm receives as inputs the aircraft rotation and the airport set. The algorithm will verify the infeasibilities in the rotation regarding transit time, airport departure and arrival capacity, and maintenance (lines 1 to 5). In line 6 the algorithm concatenates all the infeasibility sets in a single set and if the latter is not empty it returns the rotation and the index of the first infeasbility. Else, if the rotation is feasible the algorithm returns an empty set and -1. Consequently, the rotation will be add to the recovery solution, the departure and arrival airport capacity is updated and the algorithm and moves to the next aircraft.

Algorithm 2: Feasibility verification

```
   Input: $\sigma_p, \mathcal{A}$
   Output: $\sigma_p, index$
1  $inf1 \leftarrow continuity(\sigma_p)$
2  $inf2 \leftarrow tt(\sigma_p)$
3  $inf3 \leftarrow dep(\sigma_p, \mathcal{A})$
4  $inf4 \leftarrow arr(\sigma_p, \mathcal{A})$
5  $inf5 \leftarrow maint(\sigma_p)$
6  $inf \leftarrow inf1 \cup inf2 \cup inf3cupinf4 \cup inf5$
7  if $inf \neq \{\}$ then
8      $index \leftarrow min(infList)$
9      return $\sigma_p, index$
10 else
11     return $[], -1$
```

Starting at the first infeasible flight in the infeasible rotation the next step of the recovery procedure consists of an algorithm that searches for each flight, of this part of the rotation, the domain where it is possible to depart and land without breaching airport departure and arrival capacity constraints. This search is done incrementally and it will allow to delay the rotation's flights. Added to the latter, and with the exception of those flights that are already subject to a disruptive delay, the algorithm also adds the option to cancel the flight as a delay valued -1. To implement this procedure, algorithm 3 receives as inputs the infeasible rotation, the airport capacity, the maximum amount of delay and the delay increment. After initializing the variables to save the rotation's flight domains, singletons and size of the search space (line 1 to 3) the algorithm loops through the rotation to compute and retrieve their respective values. The flight domain is initialized in line 5 and if it has been disrupted its domain assumes a single value of zero thus becoming a singleton (line 7). If the singleton makes the recovery infeasible by breaking the airport capacity constraint it will be added to the singleton list and the algorithm loops to the next flight in the rotation (line 8 and 9). If the flight departs outside the recovery the algorithm will simply return the flight ranges, the singleton list and the total number of combinations. If the flight is neither disrupted or departs outside the recovery time window then the algorithm adds the cancellation option. Afterwards it will loop incrementally through the range of delay values starting at zero and add any of them that make the flight feasible (lines 14 to 20). In lines 21 and 22 the algorithm adds the domain to the flight domains dictionary and updates the number of combinations.

Algorithm 3: Find flight domains

```
Input: $\sigma_p, \mathcal{A}, maxDelay, delayIncrement$
Output: $flightDomains, singletonList, totalCombos$
1  $flightRanges \leftarrow \{\}$
2  $singletonList \leftarrow \{\}$
3  $totalCombos \leftarrow 1$
4  for $\sigma_p(i)$ in $\sigma_p$ do
5      $domain \leftarrow \{\}$
6      if $\sigma_p(i) \cap \mathcal{D}_d \neq \{\}$ then
7          $domain \leftarrow domain \cup \{0\}$
8          $checkSingleton(\sigma_p(i), \mathcal{A}, singletonList)$
9          continue
10     if $(\sigma_p^d(i) < RTW_s) \vee (\sigma_p^d(i) > RTW_e)$ then
11         return $flightRanges, singletonList, totalCombos$
12     $domain \leftarrow domain \cup \{-1\}$
13     for $delay$ in $range(0, maxDelay, delayIncrement)$ do
14         $dep \leftarrow \sigma_p^d(i)$
15         $arr \leftarrow \sigma_p^a(i)$
16         $dep \leftarrow dep + delay$
17         $arr \leftarrow arr + delay$
18         $\sigma'_p(i) \leftarrow \sigma_p(i, dep, arr)$
19         if $(dep(\sigma'_p(i), \mathcal{A}) = []) \wedge (arr(\sigma'_p(i), \mathcal{A}) = [])$ then
20             $domain \leftarrow domain \cup delay$
21     $flightDomains[\sigma_p(i)] \leftarrow domain$
22     $totalCombos \leftarrow totalCombos * |domain|$
23 return $flightRanges, singletonList, totalCombos$
```

Each flight domain will be coded in a dictionary, using the flight number and date as the key, and a vector with all the possible delay as the value:

```
{'286802/03/08': [-1, 0], '720201/03/08': [-1, 60, 120, 180, 240, 360, 420], '720701/03/08':
[-1, 0, 120, 180, 360, 420], '730801/03/08': [-1, 0, 240, 300], '737502/03/08': [-1,
0, 60, 120, 180, 240], '742002/03/08': [-1, 0, 120], '744002/03/08': [-1, 0]}
```

Each vector consists of the domain values that the flight can assume, and the search space is obtained by computing the Cartesian product between all the vectors. Any infeasible rotation has a search space that consists of a matrix whose columns are the flights and the rows are the possible values each flight can have.

$$\begin{bmatrix} -1 & -1 & -1 & ... & -1 & -1 & -1 \\ -1 & -1 & -1 & ... & -1 & -1 & 0 \\ -1 & -1 & -1 & ... & -1 & 0 & -1 \\ ... \\ 420 & 420 & 300 & ... & 120 & 180 & 0 \\ 420 & 420 & 300 & ... & 120 & 240 & -1 \\ 420 & 420 & 300 & ... & 120 & 240 & 0 \end{bmatrix}$$

In our experiments we found that the number of rows varies between the order of magnitude 10^3 to 10^{12}, hence it is not possible a unique approach to find feasible solutions. To tackle the two distinct situations the algorithm uses a lower heuristic for the lower bound of the search space and an upper heuristic to handle the upper bound of the search space.
The lower bound is initialized at 4×10^4 by default, and the lower heuristic loops through every row of the of the matrix in order to find the optimal solutions that minimize the number of cancelled flights and the total amount of delay. Algorithm 4 receives as inputs the rotation, the index of the first infeasibility and the search space in the form of a matrix. The algorithm will recover the infeasible rotation and will output the recovered one. It starts by initializing the best solution in line 1 and henceforward it will loop through the search space to find its value. In lines 3 and 4 for the algorithm sums the the cancelled flights and the total amount of delay for each row of the matrix and in line 5 it assigns them to the new solution. The algorithm compares the new solution with the best one and if the new one is not better it will iterate (lines 6 to 10). Based on the values coded in the row, in line 11 the algorithm creates the new rotation by cancelling or delaying flights. Afterwards if the row creates a new feasible solution the best solution is updated (lines 12 to 18). After traversing the entire search space the algorithm updates the aircraft rotation with the optimal solution and returns it (lines 19 and 20).

```
Algorithm 4: Lower heuristic
   Input: $\sigma_p, index, matrix$
   Output: $\sigma'_p$
 1 $bestSol \leftarrow \{\}$
 2 for $row\ in\ matrix$ do
 3     $noCancel \leftarrow \sum_i row(i)$, if $row(i) = -1$;
 4     $totalDelay \leftarrow \sum_i row(i)$, if $row(i) \neq -1$
 5     $newSol \leftarrow \{noCancel, totalDelay, row\}$
 6     if $bestSol \neq \{\}$ then
 7         if $newSol[0] < bestSol[0]$ then
 8             continue
 9         if $(newSol[0] = bestSol[0]) \wedge (newSol[1] > bestSol[1])$ then
10             continue
11     $\sigma'_p \leftarrow \sigma_p(row)$
12     if $continuity(\sigma'_p) \neq \{\}$ then
13         continue
14     if $tt(\sigma'_p) \neq \{\}$ then
15         continue
16     if $maint(\sigma'_p) \neq \{\}$ then
17         continue
18     $bestSol \leftarrow newSol$
19 $\sigma'_p \leftarrow \sigma_p(bestSol[2])$
20 return $\sigma'_p$
```

For the initialization starts The upper heuristic's upper bound is initialized at 3×10^{12} by default. This heuristic decomposes the rotation into sub-rotations with a search space size lower than the lower bound defined for the lower heuristic. The algorithm loops through every sub-rotation until it finds a feasible solution for the entire rotation. Although this procedure does not return an optimal solution, it can find feasible solutions in a reasonable computing time. Algorithm 5 receives as inputs the infeasible rotation, the index of the first infeasibility and flight domains. On line 1, the lower index of the sub-rotation is initialized with the value of the index of the first infeasibility. I line 2 the algorithm uses a sub-routine that computes the upper index for the sub-rotation that will be recovered and extracts the corresponding flight domains. In line 3 the algorithm computes the search space and in line 4 reduces the flight domains to the un-recovered part of the inputted rotation. From line 5 to 29 the algorithm will loop through every sub-rotation, recovering each one of them. With the exception of maintenance check, the algorithm finds the best solution to recover the sub-rotation using a similar method as in the lower heuristic (lines 7 to 21). In line 22 the part between the lower and the upper index of the inputted rotation is updated with the best solution. If the upper index equals the size of the inputted rotation this means the recovery procedure is over and it returns the recovered rotation (lines 23 and 24). Else, the algorithm assigns the upper index to the lower index, computes the new upper index and new partial flight domains, computes the search space and the remaining flight domains.

Algorithm 5: Upper heuristic

```
Input: $\sigma_p, index, flightDomains$
Output: $\sigma_p$
1  $lIndex \leftarrow index$
2  $uIndex, partialFlightDomains \leftarrow$
     $upperIndex(lIndex, \sigma_p, flightDomains)$
3  $matrix \leftarrow product(partialFlightDomains.values())$
4  $removeFlightDomains(flightDomains, \sigma_p(i) \forall i \in [uIndex, |\sigma_p|])$
5  while $True$ do
6      $bestSol \leftarrow \{\}$
7      for $row$ in $matrix$ do
8          $noCancel \leftarrow \sum_i row(i)$, if $row(i) = -1$;
9          $totalDelay \leftarrow \sum_i row(i)$, if $row(i) \neq -1$
10         $newSol \leftarrow [noCancel, totalDelay, row]$
11         if $bestSol \neq \{\}$ then
12             if $newSol[0] < bestSol[0]$ then
13                 continue
14             if $(newSol[0] = bestSol[0]) \wedge (newSol[1] > bestSol[1])$ then
15                 continue
16         $\sigma'_p(i) \leftarrow \sigma_p(row)$
17         if $continuity(\sigma'_p) \neq \{\}$ then
18             continue
19         if $tt(\sigma'_p) \neq \{\}$ then
20             continue
21         $bestSol \leftarrow newSol$
22     $\sigma_p = newPartialRotation(bestSol[2], \sigma_p(i) \forall i \in [lIndex : uIndex])$
23     if $uIndex = |\sigma_p|$ then
24         return $\sigma_p$
25     else
26         $lIndex \leftarrow uIndex$
27         $uIndex, partialFlightDomains \leftarrow$
             $upperIndex(lIndex, \sigma_p, flightDomains)$
28         $matrix \leftarrow product(partialFlightDomains.values())$
29         $removeFlightDomains(flightDomains, \sigma_p(i) \forall i \in$
             $[uIndex, |\sigma_p|])$
```

It is important to notice that in order to optimize the looping through the search space, the heuristics compare the current solution with the new one and if the latter is not better they will not proceed to test feasibility, thus saving significant computation time. As for the over arching algorithm, in every iteration

it loops through the aircraft list and based on the size of the search space decides which heuristic will find solutions to recover the infeasible rotations, minimizing the number of cancelled flights and the total amount of delay. However, the search space size may be above the lower bound or below the upper bound, hence the infeasible rotation is not recovered. To overcome this situation the algorithm iterates the aircraft loop using the list of aircraft left with infeasible rotations, increments the lower bound and decrements the upper bound.This procedure, combining the movement of the lower and upper bounds, results in a pincer movement that will entrap the entire search space, making sure that every infeasible rotation recovers.
In CSP it is common to find variables that have domain size 1, such variables are designated by singletons. Since both heuristics are based in constraint satisfaction programming, the rotations that we know in advance having variables with domain size 1, are handled first. In case of rotations with schedule maintenance, the algorithm treats them as a flight without turn round time and with the same origin and destination. Thus when the algorithm creates the aircraft list, the first aircraft have scheduled maintenance. The flights that are disrupted with delays are designated by fixed flights and they too cannot be moved. In this case the domain is a singleton consisting of value $\{0\}$ and if this value is infeasible, because there are no available departure and/or arrival airport capacity, the algorithm backtracks by removing the rotation of an aircraft that can release the necessary airport capacity.

Algorithm 6 receives as inputs the singleton list, the airport capacity, the ARP current solution, the rotation nad the index of the first infeasibility. The algorithm will return the ARP solution without the rotation that will allow the singleton to become feasible and, the updated aircraft list. This algorithm will loop while the singleton list is not empty and will verify if the first singleton's infeasibility is on the departure and/or in the arrival airport capacity. Depending on the situation the algorithm computes the airport time slot for the departure/arrival, and based on the origin/destination airport searches the aircraft to cancel. The algorithm will then remove the aircraft from the solution list, and the respective rotation from the ARP solution. Finally, we will use algorithm 3 to determine there are any more infeasible singletons.

Algorithm 6: Backtracking

```
Input: $singletonList, \mathcal{A}, aircraftSolList, solutionARP, \sigma_p, index$
Output: $solutionARP, aircraftSolList$
1  while $singletonList \neq \{\}$ do
2      if $singletonList(0) =' dep'$ then
3          $startInt \leftarrow 60 * int(singleton^d(0)/60)$
4          $endInt \leftarrow startInt + 60$
5          $origin \leftarrow singleton^o(0)$
6          $flight2Cancel \leftarrow solutionARP[(origin, startInt, endInt)]$
7          $airc2Cancel \leftarrow updateMulti(flight2Cancel, \mathcal{A}, solutionARP])$
8          $aircraftSolList \leftarrow aircraftSolList - airc2Cancel$
9          $solutionARP.pop(airc2Cancel)$
10         $flightRanges, singletonList, totalCombos \leftarrow$
             $domainFlights(\sigma_p(i)\forall i \in [index, |\sigma_p|], \mathcal{A}, index)$
11     if $singleton(0) =' arr'$ then
12         $startInt \leftarrow 60 * int(singleton^a(0)/60)$
13         $endInt \leftarrow startInt + 60$
14         $destination \leftarrow singleton^f(0)$
15         $flight2Cancel \leftarrow solutionARP[(destination, startInt, endInt)]$
16         $airc2Cancel \leftarrow updateMulti(flight2Cancel, \mathcal{A}, solutionARP)$
17         $aircraftSolList \leftarrow aircraftSolList - airc2Cancel$
18         $solutionARP.pop(airc2Cancel)$
19         $flightRanges, singletonList, totalCombos \leftarrow$
             $domainFlights(\sigma_p(i)\forall i \in [index, |\sigma_p|], \mathcal{A}, index)$
20 return $solutionARP, aircraftSolList$
```

After finding a feasible solution for the part of the rotation that was initially infeasible the algorithm tries to reconnect both parts but on occasions it is possible to find discontinuities between them. Algorithm 33 receives as inputs the set of aircraft disruptions, the set of distances, the origin airport, the recovered rotation the maximum flight number and returns the updated recovered rotation and the maximum flight number. Algorithm 33 will either create a taxi flight to connect the first part of the rotation and the recovered one, or cancel flights in the recovered rotation until the continuity infeasibility is removed.

It starts in line 1 initializing the origin slots where there is available departure capacity. Afterwards the algorithm will loop through the flights of the recovered rotation and if their origin is the same as origin airport it returns the updated recovered rotation and the maximum flight number (lines 2 to 4). In line 5 the algorithm computes the distance from the origin airport and the origin of the flight in the recovered rotation. In line 6 the algorithm extracts the upper slots from the origin slots by subtracting to the flight's departure time in the recovered rotation the distance and the turn round time. In line 7 the upper slots and the aircraft breakdown period are subtracted to the origin slots in order to retrieve the lower slots from where the aircraft can depart. If there are no available departure slots the algorithm cancels the flight in the recovered rotation and continues to the next flight (lines 8 to 10). If there are available departure slots at the origin, the algorithm tries to find destination slots with available airport arrival capacity. To achieve the latter the algorithm finds the destination slots with available capacity, the upper destination slots, and by subtracting the latter to the former and aircraft breakdown period determines the lower destination slots (lines 11 to 13). If there are no lower destination slots, the the algorithm cancels the flight in the recovered rotation and continues to the next flight. Since $\mathcal{A}$ is a dictionary it is necessary to extract the destination intervals and initialize the destination index i and the the time offset from which the flight departs and arrives in feasible airport slots (lines 17 to 21). The algorithm will then loop through the destination slots and in line 23 it will initialize the object obj with the starting and end time of the origin lower slots plus the distance and, in line 24 determine the index of intersection in the destination slot and the offset. If the index i is different from -1 the algorithm will the taxi flight and add it to the recovered rotation (lines 25 to 33).

Algorithm 7: Taxi flights

```
Input: $\mathcal{B}, \Delta, originAirport, \sigma_p, \mathcal{A}, M$
Output: $\sigma_p, M$
1  $originSlots \leftarrow \mathcal{A}[originAirport]$ if $capDep > noDep$
2  for $\sigma_p(i)$ $in$ $\sigma_p$ do
3      if $\sigma_p^o(i) = originAirport$ then
4          return $\sigma_p$, M
5      $\delta_{of} \leftarrow \Delta(originAirport, \sigma_p^o(i))$
6      $originSlotsUpper \leftarrow originSlots$ if $endInt > \sigma_p^d(i) - \delta_{of} - t_{rp}$
7      $originSlotsLower \leftarrow originSlots - originSlotsUpper - [\mathcal{B}_p^s, \mathcal{B}_p^e]$
8      if $originSlotsLower = \{\}$ then
9          cancel($\sigma_p(i)$)
10         continue
11     $destinationSlots \leftarrow \mathcal{A}[\sigma_p^o]$ if $capArr > noArr$
12     $destinationSlotsUpper \leftarrow destinationSlots$ if $endInt > \sigma_p^d(i) - t_{rp}$
13     $destinationSlotsLower \leftarrow$
         $destinationSlots - destinationSlotsUpper - [\mathcal{B}_p^s, \mathcal{B}_p^e]$
14     if $destinationSlotsLower = \{\}$ then
15         cancel($\sigma_p(i)$)
16         continue
17     $destIntervals \leftarrow \{\}$
18     $i \leftarrow -1$
19     $offset \leftarrow -1$
20     for $x \in destinationSlotsLower$ do
21         $destIntervals \leftarrow destIntervals \cup [x^s, x^e]$
22     for $os(i)$ $in$ $originSlotsLower$ do
23         $obj \leftarrow interval(os^s, os^e) + distInitRot$
24         $i, offset \leftarrow obj.findIntersection(destIntervals)$
25         if $i \neq -1$ then
26             $taxiFlight^o \leftarrow originAirport$
27             $taxiFlight^d \leftarrow os['startInt'] + offset$
28             $taxiFlight^f = \sigma_p^o(i)$
29             $taxiFlight^a = taxiFlight^d + \delta_{of}$
30             $taxiFlight['flight'] = M$
31             $M \leftarrow M + 1$
32             $\sigma_p \leftarrow \sigma_p + taxiFlight$
33             return $\sigma_p, M$
```