Cardinality Reasoning for bin-packing constraint. Application to a tank allocation problem

Pierre Schaus[1], Jean-Charles Régin[2], Rowan Van Schaeren[3], Wout Dullaert[4], and Birger Raa[5]
[1] ICTEAM, Université catholique de Louvain, Belgium, pschaus@gmail.com
[2] University of Nice, gcregin@gmail.com
[3] Antwerp Maritime Academy, rowan.van.schaeren@hzs.be
[4] VU University Amsterdam and University of Antwerp, wout.dullaert@vu.nl
[5] University of Gent, birger.raa@ugent.be

Abstract. Flow reasoning has been successfully used in CP for more than a decade. It was originally introduced by Régin in the well-known Alldifferent and Global Cardinality Constraint (GCC) available in most of the CP solvers. The BinPacking constraint was introduced by Shaw and mainly uses an independent knapsack reasoning in each bin to filter the possible bins for each item. This paper considers the use of a cardinality/flow reasoning for improving the filtering of a bin-packing constraint. The idea is to use a GCC as a redundant constraint to the BinPacking that will count the number of items placed in each bin. The cardinality variables of the GCC are then dynamically updated during the propagation. The cardinality reasoning of the redundant GCC makes deductions that the bin-packing constraint cannot see since the placement of all items into every bin is considered at once rather than for each bin individually. This is particularly well suited when a minimum loading in each bin is specified in advance. We apply this idea on a Tank Allocation Problem (TAP). We detail our CP model and give experimental results on a real-life instance demonstrating the added value of the cardinality reasoning for the bin-packing constraint.
Keywords: Tank Allocation, Constraint Programming, Load Planning

1 Bin-Packing Constraint

The BinPacking constraint was introduced in [1]:

$$BinPacking([X_1, \ldots, X_n], [w_1, \ldots, w_n], [L_1, \ldots, L_m]).$$

This constraint enforces the relation $L_j = \sum_i (X_i = j) \cdot w_i, \forall j$. It makes the link between n weighted items (item i has a weight w_i) and the m different capacitated bins in which they are to be put. Only the weights of the items are integers, the other arguments of the constraints are finite domain (f.d.) variables. Note that in this formulation, Lj is a variable which is bounded by the maximal capacity of the bin j. Without loss of generality we assume the item variables and their weights are sorted such that $w_i \leq w_{i+1}$. Example: $BinPacking([1, 4, 1, 2, 2], [2, 3, 3, 3, 4], [5, 7, 0, 3])$.

Classical formulation The traditional way to model a BinPacking constraint is to introduce a binary variable $B_{i,j}$ for each pair (item, bin) which is 1 (true) if item i is placed into bin j, 0 (false) otherwise. Then for each bin j, we add the constraint $L_j = \sum_i B_{i,j} \cdot w_i$. As noted in [1] one important redundant constraint to add is $\sum_j L_j = \sum_i w_i$ allowing a better communication between the other constraints.

Existing Filtering Algorithms A specific filtering algorithm for the BinPacking constraint in addition to its classical formulation has been first proposed in [1]. This algorithm essentially filters the domains of the X_i's using a knapsack-like reasoning to detect if forcing an item into a particular bin j would make it impossible to reach a load L_j for that bin. This procedure is very efficient but can return false positive saying that an item is OK for a particular bin while it is not. Shaw [1] also introduced a failure detection algorithm computing a lower bound on the number of bins necessary to complete the partial solution. This last consistency check has been extended by [2]. Finally, Cambazard and O'Sullivan [3] propose to filter the domains using an LP arc-flow formulation.

The existing filtering algorithms use the upper bounds of the loading variables $\max(L_j)$ (i.e. capacity of the bins). They do not focus much on the lower bounds of these variables $\min(L_j)$. In classical bin-packing problems, the capacity of the bins $\max(L_j)$ are constrained while the lower bounds $\min(L_j)$ are usually set to 0 in the model. The additional cardinality/flow based filtering we introduce is well suited when those lower bounds are also constrained initially $\min(L_j) > 0$.

2 Cardinality reasoning for bin-packing

Existing filtering algorithms for the *BinPacking* constraint do not make use of the cardinality information inside each bin (*i.e.*. the number of items packed inside each bin). However this information can be very valuable in some situations. Consider the extreme case where every item has an equal weight (assume a weight of 1) such that the *BinPacking* constraint reduces to a *GCC*. It is clear that the filtering algorithms for the *BinPacking* are very weak compared to the global arc consistent filtering for the GCC in such a situation. Of course this situation rarely happens in practice but in many applications the weights of the items to place are not so different and it is preferable not to lose completely the reasoning offered by a cardinality constraint (the flow reasoning). Our idea is to introduce one redundant *GCC* in the modelling of the *BinPacking*:

$$GCC([X_1, \ldots, X_n], [C_1, \ldots, C_m])$$

with C_j a variable that represents the number of items placed into bin j. Initially $Dom(C_j) = \{0, \ldots, n\}$. The cardinality variables are pruned dynamically during the search as the bounds of the L_j's and the domains of the X_i's change. Let $bound(X_i, j)$ be equal to true if the variable X_i is instantiated to value j (*i.e.* item i is placed into bin j), false otherwise. Let l_j be the load of the packed items into bin j: $l_j = \sum_{i:bound(X_i,j)} w_i$ and c_j be the number of packed items into bin

j: $c_j = \sum_{i:bound(X_i,j)} 1$. Note that it is possible that $\min(L_j) \geq l_j$ because of the filtering from [1]. Furthermore let $poss_j$ be the set of possible items into bin j: $poss_j = \{i \mid |Dom(X_i)| > 1 \ \wedge \ j \in Dom(X_i)\}$. Given a subset of items S , let $sum(S) = \sum_{i \in S} w_i$. The rules to update the lower and upper bounds of C_j are obtained by combining cardinalities and capacity information:

$$\min(C_j) \leftarrow \max(\min(C_j), c_j + |A_j|) \tag{1}$$

$$\max(C_j) \leftarrow \min(\max(C_j), c_j + |B_j|) \tag{2}$$

where $A_j \subseteq poss_j$ is the minimum cardinality set of items such that $l_j + sum(A_j) \geq \min(L_j)$ and $B_j \subseteq poss_j$ is the maximum cardinality set of items such that $l_j + sum(B_j) \leq \max(L_j)$. Since items $w_1, \ldots, w_n$ are sorted increasingly, both rules (1) and (2) can be implemented in $O(n)$ by scanning the items from right to left for rule (1) and from left to right for rule (2).

Example 1. Five items with weights 3,3,4,5,7 can be placed into bin 1 having a possible load $L_1 \in [20..22]$. Two other items are already packed into that bin with weights 3 and 7 ($c_1 = 2$ and $l_1 = 10$). Clearly we have that $|A_1| = 2$ obtained with weights 5,7 and $|B_1| = 3$ obtained with weights 3,3,4. The domain of the cardinality variable C_1 is thus set to $[4..5]$.

3 Tank allocation for liquid bulk vessels

The tank allocation problem involves the assignment of different cargoes (volumes of chemical products to be shipped by the vessel) to the available tanks of the vessel. The loading plans of bulk vessels are generally generated manually by the vessel planners although it is difficult to generate high quality solutions. The constraints to satisfy are mainly segregation constraints:

1. prevent chemicals from being loaded into certain types of tanks because
   - the chemical may need to have its temperature managed and the tank needs to be equipped with a heating system,
   - the tank must be resistant to the chemical,
   - a tank may still be contaminated by previous cargoes incompatible with the chemical.
2. prevent some pairs of cargoes to be placed next to each other: not only the chemical interactions between the different cargoes need to be considered but also the temperature at which they need to be transported. Too different temperature requirements for adjacent tanks cause the second one to solidify due to cooling off by the first cargo or the first may become chemically unstable due to heating up of the second cargo.

In order to minimize the costs and inconvenience of tank cleaning, an ideal loading plan should maximize the total volume of unused tanks (i.e. free space).

Instance The characteristics of the real instance[1] that we received from a major chemical tanker company:

- 20 cargoes with volumes ranging from 381 to 1527 tons.
- The vessel has 34 tanks with capacities from 316 to 1017 tons.
- There are 5 pairs of cargoes that cannot be placed into adjacent tanks.
- Each tank has between 1 to 3 cargoes that cannot be assigned to it.

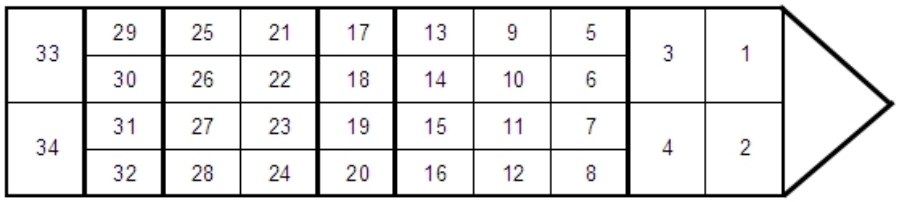

Fig. 1. Layout of the vessel.

4 A CP Model

The whole tank allocation problem is a mixed integer programming problem since the decision of which cargo is assigned to each tank is discrete but the exact volume to assign to each tank is a continuous decision. This paper only deals with the discrete problem by assigning each cargo to a set of tanks having a total capacity large enough to accommodate the whole cargo volume. The subsequent decision of the distribution of the cargo volume among those tanks must take the stability constraints into account and is beyond the scope of this paper. We just assume here that all cargo must be completely loaded. The Scala model of the problem implemented in OscaR [4] is given in Listing 1.1. In this model, two sets of variables are introduced:

- $cargo_t$: represents the type of cargo assigned to cargo tank t (type 0 represents the empty cargo). The domain of $cargo_t$ only contains cargo identifiers that can be placed into that specific cargo tank (remember that not every tank can accommodate every cargo).
- $load_c$: represents the total tank capacity available for shipping cargo c. The minimum value of $load_c$ is set to the total volume of cargo c: $volume_c$. For $load_0$ the minimum is set to 0 since there is no need to have empty cargo tanks.

[1] available upon request

Listing 1.1. Scala/OscaR CP Model

```
class Cargo(val volume: Int)
class Tank(val id: Int, val capa: Int, val neighbors: Set[Int], val possibleCargos: Set[Int])
val cargos: Array[Cargo] // all the cargo data
val tanks: Array[Tanks] // all the tanks data
val compatibles: Set[(Int, Int)] // compatibles neighbor cargos

val cp = CPSolver()
// the cargo type for each tank (dummy if empty tank)
val cargo = Array.tabulate(tanks.length)(t => CPVarInt(cp, tanks(t).possibleCargos))
// the total capacity allocated to cargo (at least the volume to place)
val load = Array.tabulate(cargos.size)(c => CPVarInt(cp, cargos(c).volume to totCapa))
// objective = the total empty space = volume allocated to dummy
val freeSpace = load(0)
// tanks allocated to cargo c in current partial solution
def tanksAllocated(c: Int) =
      (0 until tanks.size). filter (t => (cargo(t).isBound && cargo(t).getValue == c))
// volume allocated to cargo c in current partial solution
def volumeAllocated(c: Int) = tanksAllocated(c).map(tanks(_).capa).sum
// the objective, the constraints and the search
cp.maximize(freeSpace) subjectTo {
    // links cargo and load vars with binpacking constraint
    cp.add(binpacking(cargo, tanks.map(_.capa), load), Strong)
    // new cardinality redundant constraints
    cp.add(binpackingCardinality(cargo, tanks.map(_.capa), load))
    // dominance rules constraints
    for (i <- 1 until cargos.size) {
      cp.add(new DominanceRules(cargos(i),tanks,cargo))
    }
    // two neighbor tanks, must contain compatible cargo types
    for (t <- tanks; t2 <- t.neighbors; if (t2 > t.id)) {
      cp.add(table(cargo(t.id-1),cargo(t2-1),compatibles))
    }
} exploration {
    while(!allBounds(cargo)) {
      val volumeLeft = Array.tabulate(cargos.size) (c => cargos(c).volume -
            volumeAllocated(c))
      // unbounds cargo with their index
      val unboundTanks = cargo.zipWithIndex.filter{case (x,c) => !x.isBound}
      // unbound cargo (and its index) with the largest capa, tie break on domain size
      val (tankVar,tank) = unboundTanks.maxBy{case (x,c) => (tanks(c).capa,-x.getSize)}
      // cargo with largest volume still to place that can be used in the selected tank
      val cargoToPlace = (0 until
            cargos.size). filter (tankVar.hasValue(_)).maxBy(volumeLeft(_))
      cp.branch(cp.post(tankVar == cargoToPlace)) // left branch
            (cp.post(tankVar != cargoToPlace)) // right branch
    }
  }
}
```

Example 2. Consider the Figure 2. The vessel is divided into four different tanks with capacities 500, 400, 640, 330. There are two cargoes to load (A and B). The quantity of cargo A to load is 1000 and of cargo B is 790. One bin is introduced for each of them with lower bounds $\min(load_A) = 1000$ and $\min(load_B) = 790$. The objective is to assign the tanks (items) to them such that this minimum load is met meaning that all the cargo volumes can be loaded into the tanks. An assignment of the tanks to the cargoes satisfying this requirement is given on the picture: $1040 \geq 1000$ and $830 \geq 790$.

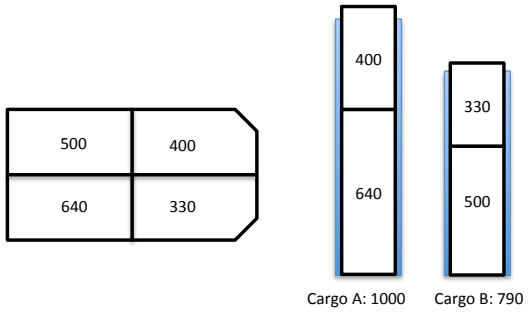

Fig. 2. Tank assignment.

Assigning cargo to tanks (or tanks to cargoes in our model), is handled with the *BinPacking* constraint to express the volume requirements of each cargo. This global constraint enforces the following relation:

$$load_c = \sum_{t=1}^{nbTanks} capa_t \cdot (cargo_t = c) \ , \forall c$$

linking the two sets of variables $cargo_t$, $load_c$ and the tank capacities $capa_t$. The segregation constraints require the layout of the cargo vessels: which cargo tanks are considered adjacent and which are not. Let $\mathcal{A} \subset [1..nbTanks] \times [1..nbTanks]$ be the set of pairs of adjacent tanks and let $\mathcal{C} \subset [1..nbCargoes] \times [1..nbCargoes]$ be the set of pairs of cargoes which are compatible. We must have that

$$(cargo_{t_i}, cargo_{t_j}) \in \mathcal{C}, \quad \forall (t_i, t_j) \in \mathcal{A}.$$

These constraints are enforced with classical table constraints. The objective function of the problem is the maximization total capacity in the unused tanks: $maximize(load_0)$.

Dominance rules Let T_c be the set of tanks allocated to a cargo c in a solution. If there exists a subset of those tanks having enough capacity to accommodate the cargo volume, then the solution is not optimal since it can be improved by allocating the subset of tanks to the cargo. More formally, a solution is not dominated if for every cargo $c \in \{1, \ldots, nbCargoes\}$, $\forall\ t' \in T_c : \sum_{t \in (T_c - t')} capa_t < volume_c$. Avoiding to generate dominated solutions can easily be achieved by implementing a dedicated propagator. As soon as the propagator realizes there

is enough capacity to accommodate a cargo, this cargo value is removed from the domain of every unbound tank variable.

Heuristic Let us define as $left_c = volume_c - \sum_{t:bound(cargo_t) \wedge cargo_t = c} capa_t$ the difference between the volume of a cargo c and the current total tank volume allocated to it. If it is positive it means that the cargo does not have enough tank volume to transport it on the vessel. If it is negative it means there is a surplus of volume for that cargo.

The variable heuristic selects the unbound variable $cargo_t$ corresponding to the tank with the largest capacity breaking ties by preferring the variable with smallest domain size.

The value heuristic tries on the left branch to assign to $cargo_t$, the cargo $c \in Dom(cargo_t)$ having the largest $left_c$ value. On the right branch, this value is removed. The idea is to use first the tanks with large capacities for the large cargo volumes, finishing down a branch with a finer granularity of tank capacities allowing more flexibility to find good feasible solutions.

Strengthening the model with lower bounds (preliminary ideas) For every cargo c, a set of tanks is allocated to it in the final solution. Let us define as $surplus_c = \sum_{t:cargo_t = c} capa_t - volume_c$ the difference between the final total tank volume allocated to a cargo c and the volume of this cargo. An interesting question is the possibility to compute a lower bound on $surplus_c$ in the final solution. A lower bound for $surplus_c$ can be found for every cargo c by solving the following sub-problem:

$$\underline{surplus}_c = \min \ (\sum_t X_t \cdot capa_t) - volume_c \tag{3}$$

$$s.t. : \ \sum_t X_t \cdot capa_t \geq volume_c \tag{4}$$

$$X_t \in \{0, 1\} \tag{5}$$

The summations in the above model are done only on the tanks that can accommodate the cargo c i.e. $\{t : c \in Dom(cargo_t)\}$. This is indeed a relaxation since a same tank can be selected for different cargo and the segregation constraints are not considered. These sub-problems can be solved with dynamic programming in pseudo-polynomial time. The resulting values can be used to strengthen the model by adding the constraints: $\forall c : \ load_c - volume_c \geq \underline{surplus}_c$ and also to compute an upper bound on the empty tanks volume: $\overline{load_0} \leq \sum_t capa_t - \sum_c (volume_c + \underline{surplus}_c)$.

5 Experimental Results

With the new redundant bin-packing cardinality constraint [2], the first feasible solution is easily found with just 28 backtracks (this solution uses all the tanks).

[2] The flow based propagator for the GCC should be used. A forward checking propagation for the GCC does not help on this problem.

Without these redundant constraints, we were not able to find any feasible solution in one hour of computation.

The best solution using the Depth First Search (DFS) branch and bound (empty space = 1811) was found after 5 minutes and 1,594,159 backtracks but it was not possible to prove its optimality.

Using a Large Neighbourhood Search (LNS) on top of our model fixing 50% of the tanks randomly from the current best solution and restarting every 1000 backtracks, we were able to find a solution with an empty space of 2296 within 3 seconds and after a dozen of restarts.

We also experimented with two MIP solvers to solve this problem:

- lp_solve was not able to find any feasible solution.
- CPLEX could find and prove the optimum after 3 seconds confirming that the solution found with CP+LNS is optimal.

We plan to extend the TAP problem and also to consider (i) the maximization of the total volume to place in the case it exceeds the capacity of the vessel (ii) the integrated routing problem of a single vessel servicing multiple ports, and (iii) the stability constraints of the vessel which are non linear. We believe that those last two constraints will make it more difficult to build a MIP model. This is the reason why we developed a CP approach.

6 Conclusion

We introduced a new additional filtering algorithm for BinPacking constraint based on cardinality reasoning to count the number of items placed in each bin. This new filtering is particularly useful when a lower bound on the capacity is specified in the bins as in the TAP problem since it can immediately deduce a minimum number of items to place inside each bin. This new filtering was experimented and showed to be crucial to solve a real-life instance of a tank allocation problem with CP.

References

1. Paul Shaw *A Constraint for Bin Packing* CP 2004: 648-662
2. Julien Dupuis, Pierre Schaus, Yves Deville *Consistency Check for the Bin Packing Constraint Revisited* CPAIOR 2010: 117-122
3. Hadrien Cambazard, Barry O'Sullivan *Propagating the Bin Packing Constraint Using Linear Programming.* CP 2010: 129-136
4. OscaR (Scala in OR) Solver: *https://bitbucket.org/oscarlib/oscar*