Declarative Programming 19/20

Luís Ressonha, nº 35003
Ruben Teimas, nº 39868

July 16, 2020

1 Introduction

We were proposed to build a *Safe packing* system.

In this systems the items are stacked on top of each other and each item has a weight(how much weight will be put on top of the next item) and strength(how much weight it can take from the previous item without being squashed) property.

For the whole system to be safe no item can have a strength property smaller than sum of the weight of the items above it.

2 Development

2.1 (A)

Our system can be represented as a *CSP*(Constraint satisfaction problem). The system, S, is given by the equation:

$$S = (V, D, C)$$

Which means the system, S, is given by a set of variables V with a domain D and respect the constraints C.

2.1.1 (A.1)

The system depends on the following **variables**:

- $\boldsymbol{N}$ = number of items to place;
- $\boldsymbol{i}$ = index of item, $\boldsymbol{i}$;
- P_i = position of item with index $\boldsymbol{i}$;
- W_i = weight of item with index $\boldsymbol{i}$;
- S_i = strength of item with index $\boldsymbol{i}$;
- X_i = item with index $\boldsymbol{i}$ which have (P_i, W_i, S_i);
- $\boldsymbol{W}$ = sum of all weight above an item, j;
- $\boldsymbol{I}$ = list of items available, $[X_0, X_1, .., X_{(N-1)}]$;

The variables have the following **domain**:

- **N,** W_i**,** $S_i \in \mathbb{N}^*$;
- ***i,*** $P_i \in [0, N-1]$;

For our model to be complete we now need to set the constraints.

We know that for the whole system to be safe all items have to be safe and an item is safe when the sum of the weight of the items above it is equal or lesser than the item's strength.

Translating it to mathematical language we have:

$$\forall X_i \in I, SS = (X_j, P_j < P_i) \subset I, \mathbf{W} \leq S_i$$

For all X with index i in list I, there is a subset SS of list I which contains the items on top of X_i and the sum of those item's weight is equal or lesser than the strength of the item X_i, just like we said before!

2.1.2 (A.2)

In order to test our *CSP* model we implemented a *CLP(FD)* program with *SWI-Prolog*. It is a very simple program in which we defined 3 predicates: *safe_pack*, *check_constraints* and *not_squashed*.

Since our indexed variables are now represented by a list, we only defined the P_i domain. The constraints were applied to the list in the *check_constraints* and *not_squashed* predicates, except for the global constraint *all_different/1* (because one item appears only once in the pack).

The solutions to the problem are lists whose elements are the index of the items in the initial given item list.

The code is listed below and can be also found in the annexed file, ***a2.pl***.

```
1  not_squashed(Wt,(_,S2)):- Wt #=< S2.
2
3  check_constraint(_,[_|[]], _).
4  check_constraint(ItemSet, [P1, P2|T], W):- nth0(P1, ItemSet, (W1,_)),
5                                              nth0(P2, ItemSet, I2),
6                                              Wt #= W+W1,
7                                              not_squashed(Wt, I2),
8                                              check_constraint(ItemSet, [P2|
                                                   T], Wt).
9
10 safe_pack(N, ItemSet, P):-    Nf is N-1,
11         length(P, N),
12         P ins 0..Nf,
13         all_different(P),
14         check_constraint(ItemSet, P, 0),
15         label(P).
```

By testing the given examples we found that there is on possible solution for the first example (N=3, WS = (5,6), (4,4), (10,10)), which is S = [1, 0, 2] or, "translated" to the corresponding items, S = [(4,4), (5,6), (10,10)].

For the second example (N=5, WS = (1,1), (2,1), (7,3), (4,4), (5,8)) there are no solutions.

2.2 (B)

In this version of the problem we were presented with 2 new variables (***M*** = *Maximum weight the pack can take;* **PO** = *number of the elements in the pack, whose domain is an integer between 0 and* **N***)* and 3 new constraints.

The first constraint is related to the new variable: this constraint states that the sum of the items weight should be equal or lesser than ***M***.

The second constraint states that the pack weight should be maximum, respecting the first constraint.

The third constraint is related to the pack occupation and the maximum weight. The pack occupation should be an integer between 0 (in case there is no solution) and N that maximizes the weight property of the pack.

By reading the constraints above it is obvious that this version of the problem is an optimization problem in which we seek to find the optimal solution to the pack's maximum weight, while being safe.

2.2.1 (B.1)

Since we are using *Prolog* to solve this problem, the simplest algorithm to tackle would be *backtracking.*

Even though this algorithm isn't particularly fast for enormous search spaces, it performs well for a relatively large set of items, like ours.

The reason why backtracking does not perform that well is because it has a starting point and will try to satisfy all the constraints, if one constraint fails with will backtrack to the point were there were no violated constraints.

Let's assume we have 3 items ((5, 6), (1,3), (2,4)). The program picks the 1^{st} item and picks the 2^{nd} item next, in this case the constraint is violated. The program would go back to the point where it picked the first item (even though there is no possible solution with the 1^{st} item on top).

Just like we said said before, this version of the problem can also be formulated as a *CSP.*

$$CSP\ P = (V, D, C)$$

In the problem above we explicitly stated all the variables off our problem, however, we will now just indicated the ones that the problem needs to find the value. This means that the variable ***N***, ***M***, ***I***, and the variables associated to it (except for P_i) will not be presented, since they all are given as input (or can be deducted from the input).

With this being said, the variables, ***V***, are:

- ***PO*** = number of items in the optimal solution;
- P_i = position of item with index ***i***;
- ***W*** = sum of all weight above an item;
- ***WT*** = total weight of the optimal solution;

The variables, $\boldsymbol{V}$ are the equivalent to an *IntVar* in a *Choco-Solver Model*.

The domain, $\boldsymbol{D}$, of each variable is:

- $\boldsymbol{PO} \in [0, \boldsymbol{N}]$;
- $P_i \in [0, \boldsymbol{(PO\text{-}1)}]$;
- $\mathbf{W}, \mathbf{WT} \in \mathbb{N}$;

The constraints of the problem's previous version also apply to this one (for the item to be safe), but for simplicity we only enumerate the ones that are specific for this version. With that being said, the constraints, $\boldsymbol{C}$, are:

- $\boldsymbol{M} \geq \boldsymbol{WT}$: The total weight of the pack can't be greater than the maximum weight that the pack can take. This constraint seems obvious, but it should be stated and respected.
- $\mathbf{MAX}(\boldsymbol{WT})$: The total weight should be maximized, respecting the previous constraints;

Given the example:

$$M = 12,\ N \leq 8,\ WS = (2,20), (3,9), (3,6), (1,8), (4,14), (5,4), (1,2), (1,7)$$

And picking, to illustrate, the item with index 2:

- M = 12;
- N = 8;
- $PO \in [0, 8]$;
- $P_2 \in [0, PO]$;
- $W_2 = 3$;
- $S_2 = 6$;
- $W \in [0, 12[$;
- $WT \in [0, 12]$;
- I = [(2,20), (3, 9), (3,6), (1,8), (4,14), (5,4), (1,2), (1,7)];

2.2.2 (B.2)

Unfortunately we couldn't make a program that would reproduce our formulation of the problem and return the solutions.

We managed to get the solutions that would weight less than $\boldsymbol{M}$ with various $\boldsymbol{PO}$ but with couldn't maximized it.

The code that we made **can be found, in an annexed file (*b2.pl)***.

2.3 (C)

The last problem is another variant of problem A, in which we wish to pack the most robust set of items.

Robustness is given by:

$$R = S_j - W$$

In which j is bottom item of the pack.

This is again an optimization problem and it's *CSP* formulation is very similar to the 1^{st} one we did for problem A with the addition of one variable, $\boldsymbol{R}$.
Following the constraints we made for problem A, for the pack to be safe, the domain of $\boldsymbol{R}$ is:

$$R \in [0; S_j - (N-1)]$$

And that is the is the variable we want to optimize.

3 Conclusion

Even though we haven't completed all the problems correctly with think we have partially succeeded.

We learned a lot about satisfaction problems, mostly because it made us look to problems from another perspective (aside from what we had with imperative programming).