The container stacking problem: a simplified constraint programming approach

Daniel Monteiro
Faculty of Engineering
University of Porto
Rua Dr. Roberto Frias, s/n 4200-465 Porto Portugal
Email: up201806185@edu.fe.up.pt

Rafael Ribeiro
Faculty of Engineering
University of Porto
Rua Dr. Roberto Frias, s/n 4200-465 Porto Portugal
Email: up201806330@edu.fe.up.pt

***Abstract*—Managing container loading and unloading operations in a container yard is an extremely challenging logistical problem. These yards always have limited capacity and individual containers may have handling constraints, not to mention plenty of other factors that may impact the planning of the yard's operations. Freight harbours (and other logistics facilities) are always looking for ways to increase their operational efficiency and to minimise their turnaround times. These kinds of challenges are very suitable to constraint programming approaches.**

The container stacking problem (CSP) reflects some of the challenges that the managers of these container yards have to contend with, and in the paper we present an extensible model to solve instances of the CSP. We also benchmark two different implementations of the model, solved with Google's OR-Tools and DOcplex (IBM's Decision Optimization CPLEX Modeling for Python).

***Index Terms*—Container stacking problem, Constraint programming.**

I. Introduction

The Container Stacking Problem, much like the Travelling Salesman Problem, features a wide range of variations, optimisation targets and unique constraints [1]. We begin by defining exactly which version of the CSP we're solving, followed by a explanation of how we model the problem. We then show how we convert a CSP instance into restrictions that we are able to insert into our model. Finally, we measure the performance of two implementations and search strategies.

II. Problem definition

Our version of the container stacking problem is based on an idealised version of a container shipping yard. We can define our version of the CSP as follows:

Given:

1) The maximum height of the stacks H
2) The number of stacks in the shipping yard S
3) The initial state of the problem (i.e. the location of the initial containers at $t = 0$)
4) The weight of the containers (heavier containers can't be stacked above lighter containers)
5) The list of shipments and gaps
   - Both shipments and gaps have a duration (by summing the duration of all the shipments and gaps, the number of states our problem has can be inferred)
   - Only shipments have a list of inbound and outbound containers. These containers need to be where they're supposed to be *before* the "ship sails"

Minimize the number of container-moving operations that are done *while* a ship is "docked".

There are four distinct operations:

- *Insert* a container into the yard
- *Remove* a container from the yard
- *Idle* (do nothing)
- *Emplace* (move a container from one stack to another)

For our implementation, we use JSON files to represent and store our CSP instances. For the example below, we wish to remove the container at the bottom of the stack. In order to make this happen, a sequence of container movements very reminiscent of the *Tower of Hanoi* problem must happen.

```
{
  "containers" : [
    ["9", 0, 0],
    ["8", 0, 1],
    ["7", 0, 2],
    ["6", 0, 3],
    ["5", 0, 4],
    ["4", 0, 5],
    ["3", 0, 6]
  ],
  "dimensions" : [3, 7],
  "shipments" : [
    {
      "duration" : 33
    },
    {
      "in": [],
      "out": ["9"],
      "duration" : 2
    }
  ],
  "weights" : {
    "9" : 9,
    "8" : 8,
    "7" : 7,
    "6" : 6,
    "5" : 5,
    "4" : 4,
    "3" : 3
  }
}
```

III. Modelling the problem

The constraints that are applied to our model can be separated into two categories: constraints related to the problem instance, and constraints not related to the problem instance. We begin with the latter category.

A. Parameters

We start with the four big dimensions of our problem:

- T : The number of states
- C : The number of containers
- S : The number of stacks
- H : The maximum height of the stacks

The indices t, c, s, h will be used to represent a value in their respective dimension. If the number of states is T, then we'll have $D = T - 1$ decisions to make. The index for D will be d.

B. Decision variables

All of the decision variables below are binary variables.

- $M_{tcsh} = 1$ if in state t, container c exists in stack s at height h
- $L_{tc} = 1$ if in state t, container c exists
- $I_{dsc} = 1$ if between states $t_1 = d$ and $t_2 = d+1$ a container was inserted into stack s at height h
- $O_{dsc} = 1$ if between states $t_1 = d$ and $t_2 = d+1$ a container was removed from stack s at height h
- $N_d = 1$ if at decision d, the decision chosen was *Insert*
- $R_d = 1$ if at decision d, the decision chosen was *Remove*
- $Z_d = 1$ if at decision d, the decision chosen was *Idle*
- $E_d = 1$ if at decision d, the decision chosen was *Emplace*

C. Constraints

Our first constraint associates M and L, ensuring that if the container is alive in L, it will also be alive in M:

$$\sum_{s=1}^{S}\sum_{h=1}^{H} M_{tcsh} = L_{tc}; t \in T; c \in C;$$

Two containers can not exist in the same place, at the same time:

$$\sum_{c=1}^{C} M_{tcsh} \leq 1; t \in T; s \in S; h \in H;$$

Containers aren't allowed to float:

$$\sum_{c=1}^{C} M_{tcsh} \leq \sum_{c=1}^{C} M_{tcs(h-1)}; t \in T; s \in S; h \in H \setminus \{1\};$$

Only one action can be chosen per decision:

$$N_d + R_d + Z_d + E_d = 1; d \in D;$$

The decision variables I and O should only be used if the corresponding decision requires it:

$$\sum_{s=1}^{S}\sum_{h=1}^{H} I_{dsh} = E_d + N_d; d \in D;$$

$$\sum_{s=1}^{S}\sum_{h=1}^{H} O_{dsh} = E_d + R_d; d \in D;$$

The number of live containers should be changed according to the action that was chosen:

$$\sum_{c=1}^{C} L_{dc} = \sum_{c=1}^{C} L_{(d+1)c} + R_d - N_d; d \in D;$$

A container can't be moved to the exact same place:

$$O_{dsh} = 0; d \in D; s \in S; h \in H; \text{Only enforce if } I_{dsh} = 1$$

If the action *remove* is chosen, make sure that O is actually pointing to an existing container:

$$\sum_{c=0}^{C} M_{dcsh} = 1; d \in D; s \in S; h \in H; \text{Only enforce if } O_{dsh} = 1$$

When a container stops existing, it won't reappear:

$$L_{dc} \geq L_{(d+1)c}; d \in D; c \in C; \text{Only enforce if } N_d = 0$$

If the action *idle* is chosen, everything stays the same:

$$M_{dcsh} = M_{(d+1)csh}; d \in D; c \in C; s \in S; h \in H;$$

$$\text{Only enforce if } Z_d = 1$$

If the action *remove* is chosen, everything stays the same, apart from one container:

$$M_{dcsh} = M_{(d+1)csh}; d \in D; c \in C; s \in S; h \in H;$$

$$\text{Only enforce if } R_d = 1 \land O_{dsh} = 0$$

If the action *emplace* is chosen and a place is empty with no I for that place, it will remain empty:

$$M_{(d+1)csh} = 0; d \in D; c \in C; s \in S; h \in H;$$

$$\text{Only enforce if } E_d = 1 \land M_{dcsh} = 0 \land I_{dsh} = 0$$

If the action *emplace* is chosen and a place is empty with an I for that place, it will **not** remain empty:

$$\sum_{c=1}^{C} M_{dcsh} = 0 \land \sum_{c=1}^{C} M_{(d+1)csh} = 1; d \in D; s \in S; h \in H;$$

$$\text{Only enforce if } E_d = 1 \land I_{dsh} = 1$$

If the action *emplace* is chosen and a place is occupied with an O for that place, that place will lose its container:

$$\sum_{c=1}^{C} M_{dcsh} = 1 \land \sum_{c=1}^{C} M_{(d+1)csh} = 0; d \in D; s \in S; h \in H;$$

$$\text{Only enforce if } E_d = 1 \land O_{dsh} = 1$$

If the action *insert* is chosen, every container that previously existed stays in the same place:

$$M_{dcsh} = M_{(d+1)csh}; d \in D; c \in C; s \in S; h \in H;$$

$$\text{Only enforce if } N_d = 1 \land I_{dsh} = 0$$

IV. Converting a problem instance into model constraints

Now that we have a solid foundation for our model, we can shift our attention towards instance-specific constraints. These constraints will be presented below in a slightly less formal manner, when compared to the previous chapter.

A. Initial state

In order to set the starting containers in their positions, we can do something like $M_{1csh} = 1$ as necessary.

B. Setting container loads and unloads limits

The information from the shipments allows us to infer how many container loads and unloads we have in our problem. With these two values (let's say X for the number of loads and Y for the number of unloads), we can implement two constraints:

$$\sum_{d=1}^{D} N_d = X \wedge \sum_{d=1}^{D} R_d = Y$$

C. Enforcing lifetime restrictions

As containers leave from and arrive to our container yard, we can be proactive in making sure that containers exist in our yard when they are supposed to, and vice versa. For example, let's imagine that we have a container that arrives with shipment 2 and departs with shipment 4. We can then enforce that the container doesn't exist in our yard up until the start of shipment two, the container exists between the end of shipment 2 and the start of shipment 4, and after the end of shipment 4 the container doesn't exist in our yard anymore. The same logic is applied to all the containers.

D. Enforcing weight restrictions

We enforced weight restrictions as follows: if a container c_1 with weight w_1 exists, any container with weight $w_c < w_1$ is not allowed to exist below c_1.

E. Minimise ship loading time

Our ultimate goal is to minimise ship turnaround time, which in turn means that we wish to maximise the amount of free time that we have during shipments. By doing this, we can shave time off our turnaround times.

Taking this into account, the goal is first and foremost to maximise idle actions during shipments, and the secondary goal is to maximise idle actions in general. By applying a weight of one thousand to the first factor, we obtained the following objective function:

Let T_1 be the subset of states that contain a docked shipment (states that are not wait times). Given $T_1 \supset T$:

$$\text{Maximize} \sum_{t_1=1}^{T_1} Z_{t_1} * 1000 + \sum_{t=1}^{T} Z_t$$

V. Implementation strategies and measuring the performance of multiple models

A. Solver Interface

Since the problem was implemented as a boolean satisfiability (SAT) constraint programming (CP) optimization problem, the criteria of choice of models with which to solve it was based on how compatible to this type of problem they were. The result of the analysis of the tools currently available resulted in two extremely compatible models: Google's OR-Tools and IBM's DOcplex, both of which provide solvers compatible with the problem formulation at hand (CP-SAT and CPLEX, respectively).

Our approach consists of using an interface ”Model” class that calls the appropriate methods for the given solver. This was possible for two reasons:

- The codebase is built entirely in Python, and thus the interface removes the need for developing two different implementations. The definition of the problem with constraints and objective function is completely separate from the it's consumption by the model.
- Solving integer programming problems in both of the libraries is done using very similar methods, such that this class was capable of providing an universal interface for them.

B. Performance

To compare both solvers, performance and scalability tests were ran on increasingly larger problems. Generating parameterizable, solvable problems proved to be very challenging for this kind of problem, so we opted for manually creating multiple cases for different tests. Constraint application and total solve times were measured.

The first round of tests, to compare performance of both solvers, evaluated the average run time over 5 rounds of solving a simple ”one ship” problem, with increasing number of containers. All other variables like number of containers in the ship that need unloading, container weight, yard size and departure time limit were kept constant. The input consists of a yard of dimensions `[6, 10]`, which translates to six stacks of height ten. The height is the amount needed for the largest problem to be considered valid, and the width is a compromise chosen in terms of making a midterm between a solvable but not too ”tight” definition. These values themselves were obtained through trial and error.

The resulting test files were named `constantContainers_{index}`.

Constraint application times were comparable on both solvers. Both constraint application and solving times grew linearly for OR-Tools and exponentially for CPLEX.

From these results, it was also clear that the scale of the problem tcitself affected performance in a larger

TABLE I: OR-Tools results

index	constraint_time (s)	solving_time (s)
0	3.23	0.29
1	6.42	0.60
2	8.92	1.07
3	12.36	1.24
4	14.45	1.62
5	19.46	1.99
6	22.55	2.34
7	24.88	2.68
8	28.18	2.47
9	29.55	1.68

TABLE II: CPLEX results

index	constraint_time (s)	solving_time (s)
0	3.31	9.27
1	5.99	30.64
2	9.76	71.49
3	12.91	232.92
4	19.90	333.55
5	23.39	445.20
6	28.18	432.88
7	29.74	500.77
8	37.33	662.25
9	34.19	433.32

factor than we expected. This is evident by the clear time difference on the problems that had a dimension above what was strictly needed (all runs but the last), which is a direct consequence of the nature of our problem formulation. This is something that was only caught in the later phase of intensive tests, and not in development.

On a broader sense, we can conclude that for this problem, CP-SAT performs much better than CPLEX. This can be explained by the nature of the problem itself, since CPLEX is better fitted to scheduling problems and CP-SAT is ideal for complex integer programming ones.

The scalabilty tests also revealed that our solution did not scale. Times in increasingly complex problems in terms of scale of input data translated into longer constraint application times, as well as longer solve times, for both solvers. Despite CPLEX suffering from this at a greater scale, with problems reaching unsolvable input conditions quicker, OR-Tools' CP-SAT also became unable to solve problems at a large enough scale.

VI. Conclusion

We are overall satisfied with what we were able to accomplish with this project both in terms of complexity and real-world relevance, despite not having explored all the options we had. We also felt that constraint programming suits the field of logistics really well.

There's always potential for improvements, and we are aware of a couple areas where our model could be expanded further, such as additional container constraint, for example. Another improvement with enormous potential would be the integration of a distance matrix into our model, which would allow us to simulate much more complex container yard interactions.

References

[1] Rekik, Ines & Elkosantini, Sabeur & Chabchoub, Habib. (2015). Container stacking problem: a survey.