Warehouse planning problem modeling

Lorenzo Bazzana
147569
bazzana.lorenzo@spes.uniud.it

April 2023

Abstract

A *planning problem* consists in finding a sequence of actions that allow to reach a determined goal i.e. a desired configuration, given a set of possible actions at each time instant. What is often desired is to reach the goal in the minimum amount of moves, optimizing the sequence of actions as to only execute the necessary moves. This report deals with the modeling of a planning problem in the *MiniZinc* and *Answer Set Programming* languages, comparing the performances of the two on a small set of possible instances.

Contents

1 Introduction

The focus of this report is to compare the performances of two modeling languages, specifically *MiniZinc*, aimed to model constraint satisfaction and constraint optimization problems, and the *Answer Set Programming* declarative programming paradigm.

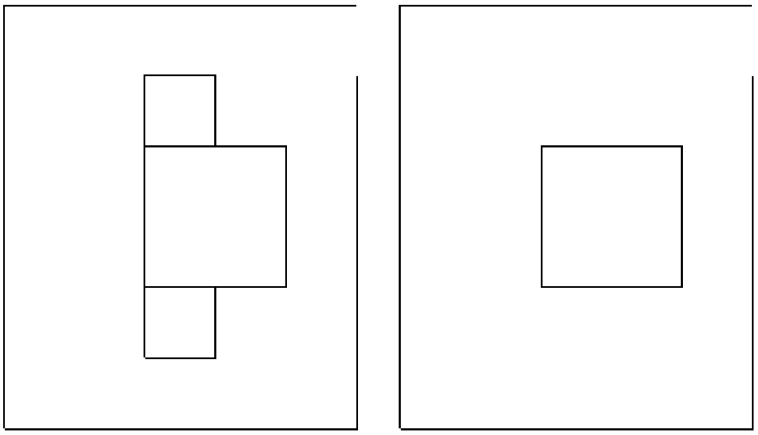


Figure 1: Instance with $m = 5$, $n = 6$, boxes in $(2, 1)$ and $(2, 4)$ and a drawer in $(2, 2)$. The right part shows the final configuration.

1.1 Problem description

The modeling problem considered is a form of warehouse planning problem, where the objective is to move all the boxes in a $m \times n$ room out of a door, placed in the top-right corner; each box can only be pushed, and not pulled: this means that in order to move a box there must be a free space behind it, relative to the direction of movement. The boxes are all of the same size, considered to be unitary. The room also contains unmovable drawers of size 2, which act as an obstacle in the way of the boxes.

The only available move is defined as $move(i, d, l)$, where:

- $i = 1, ..., k$ identifies the box to be moved
- $d = no,\ so,\ we,\ ea$ is the direction of the movement
- $l > 0$ is the length of the movement along direction d

In order to distinguish the action performed at each time step, the modeling presented in this report also includes the time instant t at which a specific action is performed, meaning that the adopted notation is $move(i, d, l, t)$.

The inputs to the problem are:

- The room size m and n
- The coordinates (x_i, y_i) of each box i, $0 \leq x \leq m - 1$ and $0 \leq y \leq n - 1$
- The coordinates (x_d, y_d) of the bottom-left corner of each drawer, $0 \leq x_d \leq m - 1$ and $0 \leq y_d \leq n - 1$

Figure 1 illustrates a possible instance for the problem.

1.2 Assumptions

Two assumptions that were adopted for the modelization of the problem are that the warehouse worker's movement need not be included in the planning (the worker can "fly" on the pushing position) and that, if there are multiple boxes aligned, they can all be pushed, proven that the pushing conditions are respected.

2 ASP modeling

The ASP model for the problem was developed with an *Action Description Language* approach in mind, meaning that the primary focus was to model the atoms that are true in a valid state in each time instant (much like the fluents for ADLs) and to define, at each time step, the valid moves that allow the transition to the next valid state.

2.1 Initial predicates

The initial part of the modeling process consists in the definition of the predicates that allow to describe the room, the valid directions and objects such as boxes and drawers.

```
1  coordX(0..m-1).
2  coordY(0..n-1).
3
4  direction(no;so;ea;we).
5
6  time(0..maxtime).
7
8  cell(X, Y) :- coordX(X), coordY(Y).
9  cell(m, n-1).
```

The code snippet shows that the valid coordinates are represented by the predicates `coordX/1` and `coordY/1`, and the time instants are analogously defined via the predicate `time/1`. The possible movement directions are represented by the `direction/1`, defined on strings. The possible positions for each box or drawer, called "cells", follow from all the possible coordinates, and are derived via the rule at line 8.

The atom `cell(m,n-1)` represents the cell that would be right out of the door, and will be used as the goal position for all boxes.

Constants such as m, n and t and the initial coordinates for boxes and drawers are instance-specific.

The initial configuration is described by the following set of rules:

```
10 boxIn(I, X, Y, 0) :- cell(X, Y), box(I, X, Y).
11
12 coveredByDrawer(X, Y) :- cell(X, Y), drawer(X, Y).
13 coveredByDrawer(X, Y) :- cell(X, Y), drawer(X-1, Y).
14 coveredByDrawer(X, Y) :- cell(X, Y), drawer(X, Y-1).
15 coveredByDrawer(X, Y) :- cell(X, Y), drawer(X-1, Y-1).
```

The `boxIn/4` predicate is an extension of `box/3`, and it includes the time instant at which a box is in a specific cell, while `coveredByDrawer/2` is true for all the cells that are covered by a drawer, since every drawer is a square of size 2; this predicate does not include the time instant because drawers are unmovable.

Having defined the positions of the boxes and of the drawers, it is useful to have a predicate describing at each time instant the cells that are not free:

```
16 notFree(X, Y, Ti) :- cell(X, Y), time(Ti),
17                      boxIn(_, X, Y, Ti), X != m.
18 notFree(X, Y, Ti) :- cell(X, Y), time(Ti),
19                      coveredByDrawer(X, Y).
```

The condition `X != m` at line 17 is used to ensure that the target cell `cell(m,n-1)` is always considered free, so that the boxes can be all pushed out of the door.

In order to later model the movement of the boxes, we now move on to defining the spatial relations between the cells, keeping in consideration the four movement directions; the following snippet shows the north/south relationships, while the east/west ones can be defined in an analogous manner:

```
20  north(X1, Y1, X2, Y2, L) :-
21          cell(X, Y),
22          north(X1, Y1, X, Y, L1),
23          north(X, Y, X2, Y2, L2),
24          L=L1+L2.
25
26  north(X1, Y1, X2, Y2, 1) :-
27          cell(X1, Y1), cell(X2, Y2),
28          X1 = X2, Y1 = Y2+1.
29
30  south(X1, Y1, X2, Y2, L) :- north(X2, Y2, X1, Y1, L).
31
32  dest(XNew, YNew, X, Y, L, no) :- north(XNew, YNew, X, Y, L).
```

The meaning of the predicate is that a box at (x_1, y_1) is north of a box at (x_2, y_2) with distance l if there exists an intermediate box at a location (x, y) that has distance to both boxes less than l (so basically by induction, with a base case of $l = 1$). Conversely, we can derive the south relationship from the north one by flipping the order of the two boxes.

The destination of a movement in a specific direction (predicate at line 32) is derived from the spatial relationships and the use of a fixed parameter ("no","so","ea","we").

2.2 Movement predicates

The movement predicates determine whether a certain box can be moved by checking different conditions depending on the type of the movement (single box versus multiple boxes). This distinction is necessary to keep the multiple boxes movement as simple as possible.

Before analyzing the code for box movements it is necessary to introduce some auxiliary predicates, that check if a path from a starting cell and a destination cell is either blocked by a box or by a drawer, and whether a cell is a pushing position for moving a box in a certain direction:

```
33  blockedByDrawer(XStart, YStart, XEnd, YEnd) :-
34          coveredByDrawer(X,Y),
35          dest(X, Y, XStart, YStart, L1, D),
36          dest(XEnd, YEnd, X, Y, L2, D),
37          dest(XEnd, YEnd, XStart, YStart,
38          L1+L2, D).
39
40  blockedPath(XStart, YStart, XEnd, YEnd, Ti) :-
41          notFree(X, Y, Ti), dest(X, Y, XStart, YStart, L1, D),
42          dest(XEnd, YEnd, X, Y, L2, D),
43          dest(XEnd, YEnd, XStart, YStart, L1+L2, D).
44
45  pushPosition(XPush, YPush, X, Y, D) :-
46          dest(X, Y, XPush, YPush, 1, D), XPush != m.
```

With these predicates it is possible to define the movement conditions:

- For single box movements, a move $move(id, dir, l, t)$ is valid if the box id is at time t on a cell (x, y) and the path from (x, y) to (x', y') is free from other boxes or drawers, with (x', y') being distant l from the starting cell in the direction dir; the pushing position for the box must also be free.
- For multiple box movements, the necessary checks are more difficult: we must check that the path from the starting position (x, y) to the destination (x', y') is free from drawers, but also that there is enough space to moved all the other boxes that are involved in the movement and are collaterally moved by the pushed box. If these boxes are k, we must then check that the path from (x', y') to $(x', y') + k$ is free from drawers. It is also necessary to check that the cell $(x', y') + k$ is not occupied, because a box in that position is not counted in the k boxes moved (as it is in fact not moved).

```
47  validMoves(I, D, L, Ti) :-
48          box(I, _, _), direction(D),
49          time(Ti),
50          Ti < maxtime,
51          not notFree(XNew, YNew, Ti),
52          boxIn(I, X, Y, Ti),
53          not notFree(XPush, YPush, Ti),
54          dest(XNew, YNew, X, Y, L, D),
55          pushPosition(XPush, YPush, X, Y, D),
56          not blockedPath(X, Y, XNew, YNew, Ti).
57
58  validMultipleMoves(I, D, L, Ti) :-
59          box(I, _, _), direction(D),
60          time(Ti),
61          Ti < maxtime,
62          boxIn(I, X, Y, Ti),
63          not notFree(XPush, YPush, Ti),
64          pushPosition(XPush, YPush, X, Y, D),
65          not blockedByDrawer(X, Y, XMax, YMax),
66          dest(X1, Y1, X, Y, L, D),
67          dest(XMax, YMax, X1, Y1, K, D),
68          not notFree(XMax, YMax, Ti),
69          LMax = L+K,
70          K = #count{I2:boxIn(I2, X2, Y2, Ti),
71                       I2 != I,
72                       dest(XMax, YMax, X2, Y2, L2, D),
73                       L2 < LMax}.
74
```

After having defined all the possible moves for a box at a time instant, one of these is nondeterministically chosen, consequently determining the updated positions for the boxes:

```
75  0{move(I, D, L, Ti):validMoves(I, D, L, Ti);
76    move(I, D, L, Ti):validMultipleMoves(I, D, L, Ti)}1 :-
77          time(Ti), Ti < maxtime.
78
79  moved(I, D, L, Ti) :- move(I, D, L, Ti).
80  moved(I, D, L, Ti) :- collateralMove(I, D, L, Ti).
81
```

```
82  collateralMove(I, D, N, Ti) :-
83          move(I2, D, L, Ti),
84          boxIn(I2, X2, Y2, Ti),
85          dest(X2Final, Y2Final, X2, Y2, L, D),
86          boxIn(I, X, Y, Ti),
87          dest(X, Y, X2, Y2, M, D),
88          dest(XNew, YNew, X, Y, N, D),
89          N = L-M+1+K,
90          K = #count{I3:boxIn(I3, X3, Y3, Ti),
91                       dest(X3, Y3, X2, Y2, L3, D), L3 < M}.
```

The calculation for the length of a collateral move for a box i is done by calculating the distance m between the box pushed and i and the number k of boxes in between: box i is moved by a length of $l - m + 1 + k$, where l is the length of the movement of the pushed box.

2.3 Box position update

For each box, its position at instant $t + 1$ is unchanged if it has not been moved or involved in a collateral move at instant t; otherwise its position is updated accordingly to the relative `move/4` or `collateralMove/4`:

```
92  boxIn(I, X, Y, Ti+1) :-
93          time(Ti),
94          Ti < maxtime,
95          boxIn(I, X, Y, Ti),
96          not move(I, _, _, Ti),
97          not collateralMove(I, _, _, Ti).
98
99  boxIn(I, XNew, YNew, Ti+1) :-
100         time(Ti),
101         Ti < maxtime,
102         cell(XNew, YNew),
103         boxIn(I, X, Y, Ti),
104         dest(XNew, YNew, X, Y, L, D),
105         moved(I, D, L, Ti).
```

2.4 Goal and domain knowledge constraints

In order to stop search early in case of trivially unsatisfiable instances (e.g. the ones where the exit is either blocked, or the boxes cannot be pushed out of the door because the pushing position is blocked by a drawer) we add the following domain knowledge constraints:

```
106 :- coveredByDrawer(m-1, n-1).
107
108 :- coveredByDrawer(m-2, n-1).
```

The goal position for each box is the cell $(m, n - 1)$; in order to determine whether a box is not in the goal position we use a predicate `unfinished/2`, adding a constraint that prohibits having a box not in the goal position by the last time instant:

```
109 unfinished(I, Ti) :-
110         time(Ti),
```

```
            not boxIn(I, X, Y, Ti),
            box(I, _, _),
            X = m,
            Y = n-1.

:- unfinished(I, maxtime), box(I, _, _).
```

It is interesting to note that these two predicates could work by directly instantiating m and $n-1$ inside the `boxIn/3` and redefining `unfinished/2` as to only work on the box id, but experimental observations noted that the current definition is slightly faster.

Another important observation is that this model does not allow to move out of the room more than one box at a time; there are some changes that could be applied in order to allow "flushing" multiple boxes, but it has been empirically observed that these changes introduce a considerable amount of computational overhead, so it has ultimately been decided to keep the behaviour as it is. In a real-world setting, this could be justified in situations where the boxes are pushed out of the room and loaded on trucks for transport, and thus need to be taken out one at a time.

Lastly, in order to get to the goal in the minimum amount of moves, we try to minimize the number of `move/4` predicates, additionally adding a constraint to remove "idle" instants where no move is performed:

```
:- time(Ti), time(Ti+1), move(_,_,_, Ti+1), not move(_,_,_,Ti), Ti < maxtime.
requiredMoves(M) :- M = #count{T:move(I, D, L, T)}.
#minimize {M:requiredMoves(M)}.
```

3 MiniZinc modeling

The MiniZinc model defined takes on a different approach than the ASP counterpart: instead of trying to derive possible valid moves at each instant and updating the configuration accordingly, it ensures that two consecutive configurations of the room satisfy a set of constraints that represent the necessary conditions that make the transition from one configuration to the next a valid one.

3.1 Data representation

Boxes, drawers and moves are each represented by an array: `boxes` is a 3-dimensional array of variables containing the position of each box, and that is indexed by the box ids, the time instant, and the coordinate (x or y), while `moves` is a variable array containing, for each time instant, the pushed box id. Since drawers cannot be moved, `drawers` is an array of fixed integers.

```
array[1..boxNumber, 0..maxtime, 0..1] of var 0..max(m,n): boxes;
array[1..drawerNumber, 0..1] of 0..max(m,n): drawers;
array[0..maxtime-1] of var 0..boxNumber: moves;
```

3.2 Movement constraints

The main movement constraint is a predicate called `movementConstraint`, defined to check the consistency of movement of a box: if a box is moved it checks that only one of the two coordinates is changed, that the traversed cells are not covered by a drawer and either the box was the one pushed, or there was another box that was pushed in the same direction:

```
4     predicate movementConstraint(1..boxNumber: box, 1..maxtime: t, 0..1: coord) =
5         let {0..1: otherCoord = (coord+1) mod 2} in
6         not moved(box, t, otherCoord) /\
7         freePath(box, boxes[box, t-1, coord], boxes[box, t, coord], t-1, coord) /\
8         (
9             (freePushPosition(box, coord, t-1) /\ moves[t-1] = box) \/
10         (
11             exists(b2 in 1..boxNumber)(
12                 b2 != box /\
13                 moves[t-1] = b2 /\
14                 precedes(b2, box, coord, t-1) /\
15                 sameDirection(b2, box, coord, t) /\
16                    fullPath(min(boxes[b2, t, coord], boxes[box, t, coord]),
17                    max(boxes[b2, t, coord], boxes[box, t, coord]), coord,
18                    boxes[box, t, otherCoord], t)
19             )
20         )
21         );
```

In this constraint, different predicates are used to make the code more readable:

- `precedes/4`: a box precedes another one in a certain movement direction
- `freePushPosition/3`: whether a box moved in a certain direction had a free pushing position
- `freePath/5`: the path from a starting to an ending point must be free from drawers
- `sameDirection/4`: whether two boxes have been moved in a certain direction.

The predicate `fullPath/5` ensures that, after a movement, there are no empty cells between the pushed box and any other box that was moved in a collateral way.

3.3 Consistency constraints

Consistency constraints are constraints necessary to keep configurations consistent, avoiding situations like boxes overlapping (line 22), going out of bounds (line 51) and preventing boxes from swapping places between two configurations (line 34); this last constraint is also necessary to correctly model movement.

```
22    constraint
23    forall(t in 0..maxtime)(
24        forall(x in 0..m)(
25            x != m ->
26                not exists(b1 in 1..boxNumber, b2 in 1..boxNumber)(
27                    b1 != b2 /\ boxes[b1, t, 0] = x /\
28                    boxes[b2, t, 0] = x /\
29                    boxes[b1, t, 1] = boxes[b2, t, 1]
30                )
31        )
32    );
33
34    constraint
```

```
    forall(t in 0..maxtime-1)(
        not exists(b1 in 1..boxNumber, b2 in 1..boxNumber)(
            b1 != b2 /\ (
                boxes[b1, t, 0] = boxes[b2, t, 0] /\
                boxes[b1, t+1, 0] = boxes[b2, t+1, 0] /\
                boxes[b1, t, 1] < boxes[b2, t, 1] /\
                boxes[b1, t+1, 1] > boxes[b2, t+1, 1]
            ) \/ (
                boxes[b1, t, 1] = boxes[b2, t, 1] /\
                boxes[b1, t+1, 1] = boxes[b2, t+1, 1] /\
                boxes[b1, t, 0] < boxes[b2, t, 0] /\
                boxes[b1, t+1, 0] > boxes[b2, t+1, 0]
            )
        )
    );

    constraint
    forall(box in 1..boxNumber)(
        forall(t in 0..maxtime)((
            boxes[box, t, 0] < m /\
            boxes[box, t, 0] >= 0 /\
            boxes[box, t, 1] >= 0 /\
            boxes[box, t, 1] < n
            ) \/ (
            boxes[box, t, 0] = m /\
            boxes[box, t, 1] = n-1)
        )
    );
```

Another consistency constraint is required to only assign actual moves to the `moves` array, because otherwise the solver could assign to the variables in the array values relative to boxes that have not actually been moved:

```
    constraint
    forall(t in 1..maxtime)(
        forall(box in 1..boxNumber)(
            moves[t-1] = box -> (moved(box, t, 0) \/ moved(box, t, 1))
        )
    );
```

3.4 Goal and domain knowledge constraints

The goal constraint for the MiniZinc model is stil defined as having all boxes in position $(m, n-1)$ at the last time instant; in order to model the same behaviour as the ASP counterpart, we also need a constraint that forbids moving more than box out of the door at one time.

```
    constraint
    forall(box in 1..boxNumber)(
        boxes[box, maxtime, 0] = m /\
        boxes[box, maxtime, 1] = n-1
    );
```

```
74
75      constraint
76      not exists(b1 in 1..boxNumber, b2 in 1..boxNumber)(
77          exists(t in 0..maxtime-1)(
78              (boxes[b1, t, 0] != m \/ boxes[b1, t, 1] != n-1) /\
79              (boxes[b1, t+1, 0] = m /\ boxes[b1, t+1, 1] = n-1) /\
80              (boxes[b2, t, 0] != m \/ boxes[b2, t, 1] != n-1) /\
81              (boxes[b2, t+1, 0] = m /\ boxes[b2, t+1, 1] = n-1) /\
82              b1 != b2
83          )
84      );
```

Next we add the domain knowledge constraints regarding the impossibility to have boxes that block the exit:

```
85      constraint
86      not exists(drawer in 1..drawerNumber)(
87          drawers[drawerNumber, 0] = m-2 /\
88          drawers[drawerNumber, 1] = n-2
89      );
90
91      constraint
92      not exists(drawer in 1..drawerNumber)(
93          drawers[drawerNumber, 0] = m-3 /\
94          drawers[drawerNumber, 1] = n-2
95      );
```

Lastly, we need to include the minimization requirements, modeled as the need to minimize the number of elements in the `moves` array that are different from 0; we also add the constraint to avoid having no moves for a time instant before actually reaching the goal.

```
96      constraint
97      not exists(t in 0..maxtime-2)(
98          moves[t+1] != 0 /\ moves[t] = 0
99      );
100
101     var int: moveNumber = count(t in 0..maxtime-1)(moves[t] != 0);
102
103     solve minimize moveNumber;
```

4 Performance comparison

This section of the report deals with the performance comparison between the ASP and MiniZinc models, using the *Clingo* solver for ASP and the MiniZinc compiler together with the *Chuffed* solver for the MiniZinc model.

The comparison is performed over 30 instances, divided in three classes of difficulty (10 easy, 15 medium and 5 hard instances), where each instance is randomly generated, with the only constraint that there are no overlapping boxes or drawers, or objects out of bounds; it should be noted however that the level of difficulty for these instances is only indicative of the expected time performance, since there is no clear separation between “easy” and “medium”, or “medium” and “hard”. We expect easy

instances to require a very short amount of running time, medium instances to run for an interval that ranges between a few seconds and possibly a few minutes of computation, and hard instances to require a lot of running time (over 5 minutes, for example).

4.1 Clingo arguments

The solver for the ASP model used in this report is Clingo, run with 4 threads to be more competitive towards the MiniZinc counterpart. By doing some experiments with different configurations, it was observed that there isn't a substantial change in the performance, so all tests have been run with the standard configuration; however, a time limit of 5 minutes (300 seconds) was set in order to stop the search for taking too long.

4.2 MiniZinc arguments

The MiniZinc model is evaluated by using the Chuffed 0.11.0 solver, which has proven to be the fastest solver for the model presented here by a large amount. The optimization level used is 1 (`-O1` option), which empirically proved to be faster that the others, probably due to the smaller preprocessing time required; the solver was run without particular solving heuristics and using only one thread. Like for the ASP counterpart, a time limit of 5 minutes was set.

4.3 Time comparison

The comparison is done by taking into account three parameters:

- The time required to find the first model
- The total computation time
- The number of models returned

Out of all these, the total computation time is probably the most important, as it is the one that shows the biggest differences between the two models; however, depending on the situation, the other two parameters could have more weight when it comes to deciding which model to adopt.

Table 1 presents the data recorded from the tests, while figures 2, 3 and 4 show a more readable comparison between the computation times and number of models found.

From the figures it is clear that MiniZinc (i.e. the Chuffed solver) is overall faster in finding the optimum, never going over the 5 minutes limit (although on instance *hard5* it goes very close to it), while Clingo exceeds the maximum time in all of the hard instances; on the other hand, Clingo seems to be faster in finding the first model over easy instances, but that quickly changes when the instances grow in difficulty, proving again the speed of MiniZinc. Lastly, Clingo finds more models in during its computations, but that could depend on the underlying search strategy used by the solvers.

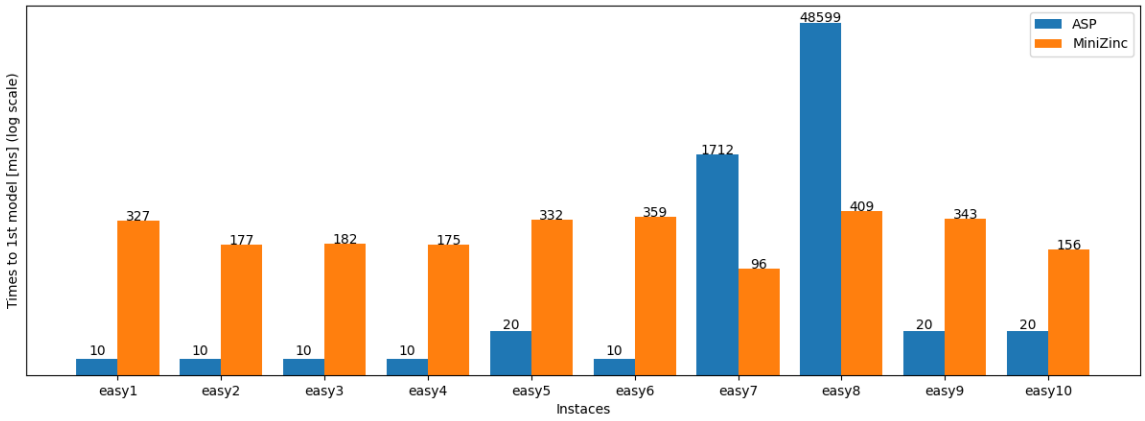
5 Conclusions

The MiniZinc model seems to be the best of the two models presented here, having outperformed the ASP version in every instance; it should be noted however that this heavily depends on the use of Chuffed 0.11.0 as the MiniZinc solver, as other solvers (for example Gecode 6.30) were observed to be much slower than Chuffed and Clingo. Another reason for the difference in performance could be intrinsic in the logical structure of the ASP model: the program calculates at each step all possible valid moves, then chooses one in a nondeterministic way and propagates that choice for all subsequent time steps. This is the simplest and probably more natural way to model the problem in the ASP language, but it is also computationally heavy, especially when the room grows in size, so an interesting future development could be to rewrite the ASP model in a similar fashion to the MiniZinc model and see if there are any positive (or negative) changes in performance.

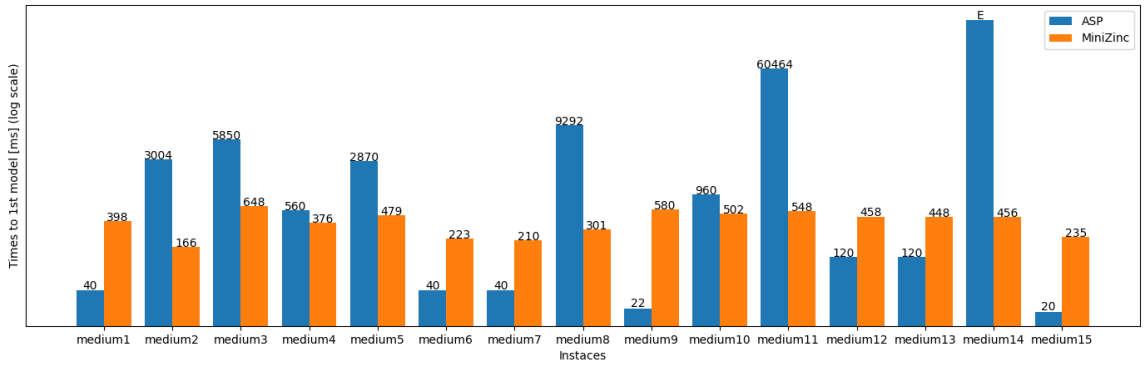
	ASP			**MiniZinc**		
Instance	1st model [ms]	Total time [ms]	N. models	1st model [ms]	Total time [ms]	N. models
easy1	10	639	4	327	356	2
easy2	10	2170	1	177	205	1
easy3	10	756	1	182	206	1
easy4	10	379	1	175	201	1
easy5	20	997	4	332	379	1
easy6	10	4403	3	359	425	1
easy7	UNSAT	1712	UNSAT	UNSAT	96	UNSAT
easy8	UNSAT	48599	UNSAT	UNSAT	409	UNSAT
easy9	20	757	3	343	395	2
easy10	20	692	2	156	181	1
medium1	40	3673	5	398	444	1
medium2	UNSAT	3004	UNSAT	UNSAT	166	UNSAT
medium3	5850	194251	3	648	1790	2
medium4	560	7999	5	376	441	2
medium5	2870	253133	1	479	775	1
medium6	40	11262	2	223	259	2
medium7	40	10978	1	210	238	1
medium8	UNSAT	9292	UNSAT	UNSAT	301	UNSAT
medium9	22	60248	4	580	670	2
medium10	960	10042	1	502	506	1
medium11	UNSAT	60464	UNSAT	UNSAT	548	UNSAT
medium12	120	14274	4	458	516	1
medium13	120	47472	4	448	496	1
medium14	E	E	0	UNSAT	456	UNSAT
medium15	20	4900	1	235	264	2
hard1	E	E	0	821	1256	3
hard2	E	E	0	3754	6200	3
hard3	3547	E	5	649	1180	2
hard4	E	E	0	797	1520	3
hard5	E	E	0	4686	285000	1

Table 1: Recorded times and number of models found.

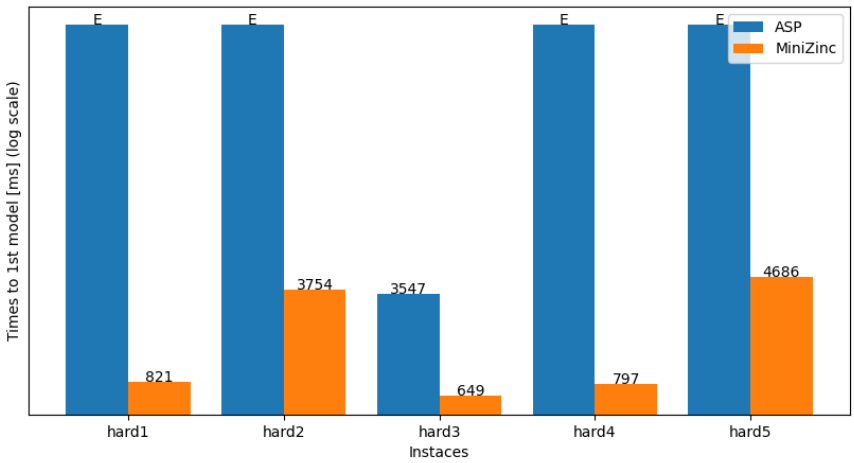
Figure 2: Computation times to find the first model.



(a) Easy instances

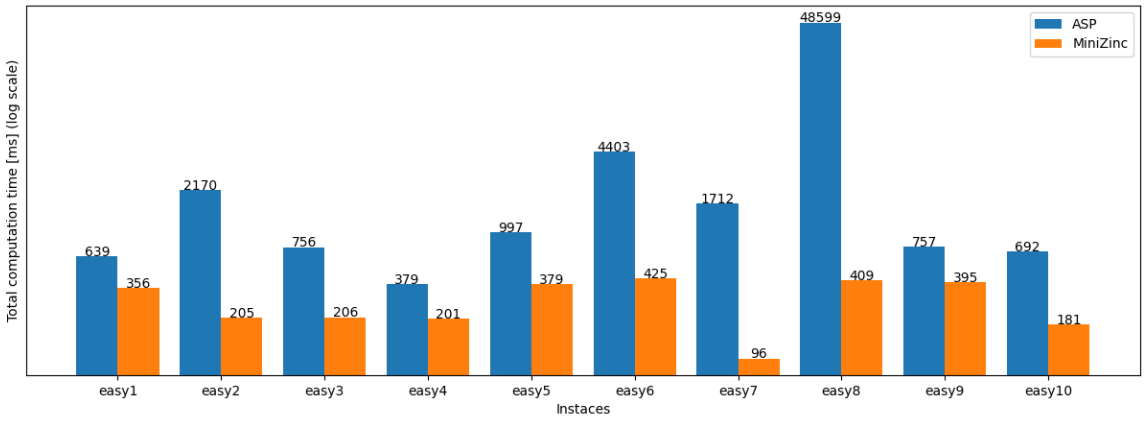


(b) Medium instances

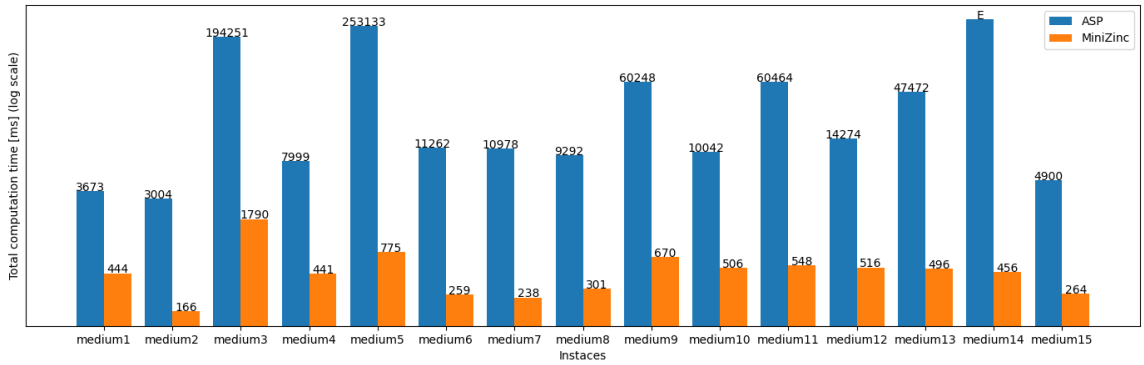


(c) Hard instances

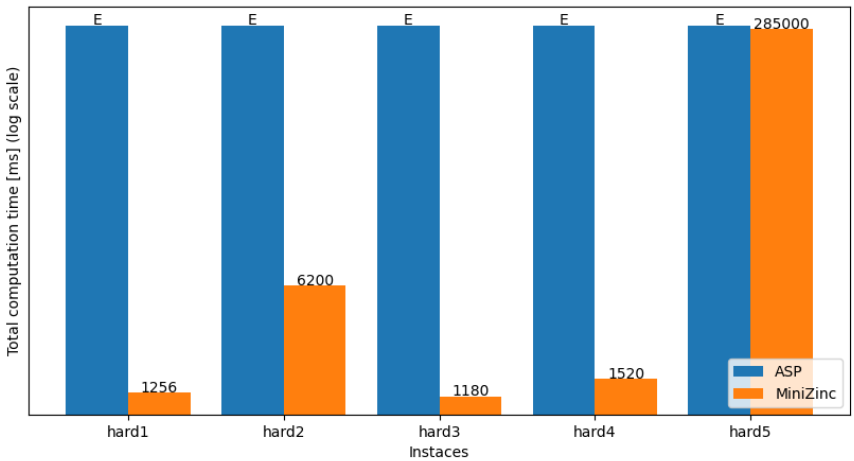
Figure 3: Total computation times (times to find the optimum model).



(a) Easy instances

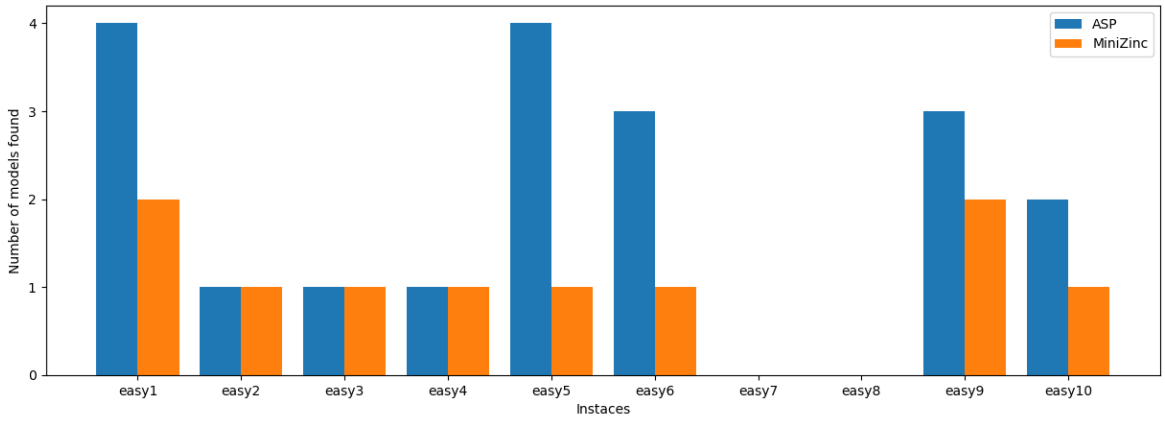


(b) Medium instances

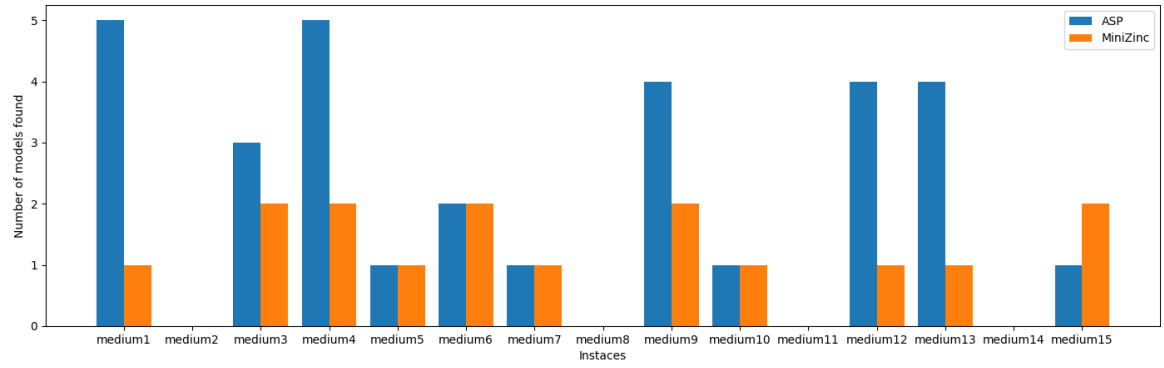


(c) Hard instances

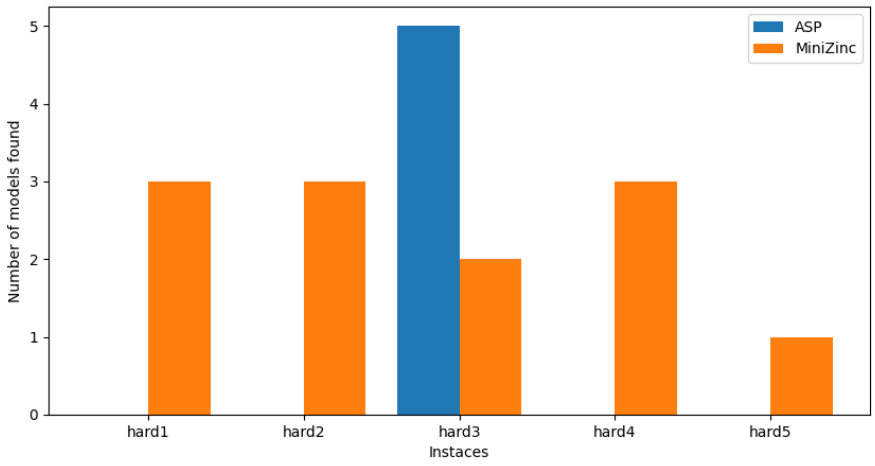
Figure 4: Number of models found.



(a) Easy instances



(b) Medium instances



(c) Hard instances