ANSWER SET AND CONSTRAINT PROGRAMMING

ZANOLIN LORENZO

Abstract

COP (Constraint Optimization Problem) are problems for which we want to find an assignment for the variables in order to satisfy some *constraints* and also to minimize a *cost function.* The aim of this report is to solve a COP problem using *Constraint Programming* and *Answer Set Programming* and compare measured performances.

Contents

1 Introduction

The objective of this activity is to compare different modelling techniques for a COP problem. Specifically, we will consider the *MiniZinc* constraint modeling language and the *ASP programming paradigm* (using the Clingo answer set solver). For each system, we will present the modeling of the general problem, showing how each constraint of the assignment can be implemented. Subsequently, we will present a Python script to generate pseudo-random instances with increasing complexity and finally we will propose some performance metrics of the two solvers on these test cases.

1.1 Problem description

The problem is similar to the famous *Sokoban Problem* [1, 2]. We need to move k boxes $b_1, \cdots, b_k$, whose size is $s_1, \cdots, s_k$, inside a $m \times n$ room. The aim is to stack all boxes at the bottom starting from the left corner using the minimum number of moves. If there is not enough space at the bottom, the worker will stack them higher until there is enough space.The only available move is *move(i,d)* which push the b_i box towards direction d.

1.1.1 Constraints

We consider two kinds of constraints:

- *Hard* Constraints: those which must be always satisfied. In this case we consider:
  - *Push*: only *pushing* is allowed, worker cannot *pull* boxes;
  - *Pushable*: to push a box, there must be *at least* an empty space in front of the box (where the worker is);
  - *Free Row*: to push a box, *all* cells beyond the box (towards decided direction d) must be empty;
  - *Inertia*: if a box is not pushed at time t_i, then it must be at the same position at time t_{i+1};
  - *Move*: only a move at time is accepted.
- *Soft* Constraints: those which aren't strictly necessary, but is better to have them satisfied:
  - *Number of moves*: number of moves has to be minimum;

An example of the problem is represented in figure 1. The left part is the initial state while the right one is the final state, in which all boxed are stacked correctly.

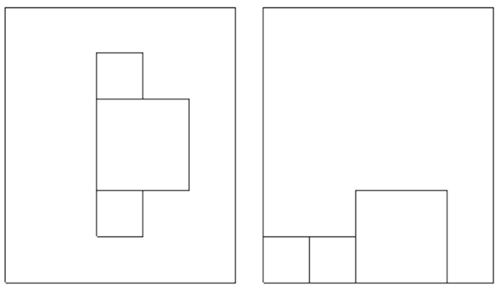

Figure 1: Initial and final state of the problem.

1.1.2 Cost function

The approach taken in the two solution is slightly different. As we will see, in the *ASP* setting we considered the *number of moves* as a function and the aim is to find global *minimum* of it; while in the *CP* one we considered the *number of null moves* as a function and the aim is to find a global *maximum* of it.

1.1.3 Assumptions

The following choices were made:

- We start stacking bigger boxes in the left corner, so they are always at the leftmost part of the room.
- For simplicity we do not consider sokoban movements, anyway we consider that the cell were the sokoban should be must be empty.

2 MiniZinc Modelling

The following *model* is composed of 4 parts, we will briefly analyze all of them.

2.1 Variables definition

We use the following variables, which are given in input from instances, to model the problem:

```
int: n;   % height of the room
int: m;   % base of the room
int: k;   % number of cartons, each of them is a square of size s_k
int: maxTime; % maximum number of steps
int: maxDim;   % maximum size of a carton
```

With the previous variables, we can the define sets that we will use as sizing for arrays.

```
set of int: STEPS = 1..maxTime-1;
set of int: CARTONS = 1..k;
set of int: ROWS = 1..n;
set of int: COLUMNS = 1..m;
set of int: SIZE = 1..maxDim;
set of int: MOVES = {0,1,2,3,4};
```

A *move* is represented using numbers, in this case: 0:DOWN 1:UP 2: LEFT 3:RIGHT 4:NONE
Finally, we use previous sets to define data structures to model the problem.

```
array [CARTONS,1..3] of par int: initialCoords;
array [CARTONS,1..2,1..maxTime] of var int: vertexCoords;
array[STEPS] of var MOVES: movesDoneAtTime;
array[STEPS] of var CARTONS: boxSelectedAtTime;
array [CARTONS,1..2] of par int: finalCoords;
array [ROWS,COLUMNS,1..maxTime] of var 0..3: room;
```

A brief description follows:

- *initialCoords* is an array of size used to contain initial coordinates of cartons, more specifically: $initialCoords[i][1]$ and $coord[i][2]$ are the initial coordinates of carton c_i , $initialCoords[i][3]$ is the length of c_i;
- *vertexCoords* is an array used to save vertex coordinates of each carton at each step, in practice it's necessary to generate successive coordinates. $vertexCoords[i][1][t]$ is the column of the cell in which box c_i stands at time t, while $vertexCoords[i][2][t]$ is the row of the cell;
- *movesDoneAtTime* is an array used to save moves done at time t;
- *boxSelectedAtTime* is used to save the box that is moved at time t, in fact it's used combined with *movesDoneAtTime*;

- *finalCoords* is used to save final coordinates for each carton and it's given in input. $finalCoords[i][1]$ is the column of the cell in which box c_i stands at time $maxTime$, while $finalCoords[i][2]$ is the row of the cell;
- *room* is used to save the configuration of the entire room at each time. $room[i][j][t]$ describe a cell at time t; in our case can be $room[i][j][t] = 0$ if the cell is empty or $room[i][j][t] = i$ if carton c_i is on it.

2.2 Predicates definition

First predicate is ***setInitialMatrix()*** which is used to instantiate correctly *room* at time $t = 0$.

```
predicate setInitialMatrix()=

  forall(row in ROWS, col in COLUMNS)(  % set empty cells
    if (not exists(c in CARTONS)(
    (row >= initialCoords[c,2] - initialCoords[c,3] + 1 /\ row <= initialCoords[c,2] ) /\
    (col >= initialCoords[c,1] /\ col <= initialCoords[c,1] + initialCoords[c,3] - 1)))
    then
         room[row,col,1] = 0
    endif
  )

  /\

  forall(c in CARTONS)(  % set occupied cells
    forall(i in 0..initialCoords[c,3]-1)(
      forall(j in 0..initialCoords[c,3]-1)(
        ( room[initialCoords[c,2]-i, initialCoords[c,1]+j,1] = c)))

  /\

   % save coordinates for each box
   vertexCoords[c,1,1] = initialCoords[c,1] /\ % x column
   vertexCoords[c,2,1] = initialCoords[c,2] % y row
);
```

In order:

- we load initial coordinates from *initialCoords*. Coordinates given describe the left-down vertex of a carton;
- we occupy all room's cells, respectively with 0 or id values. This operation is done calculating for each carton c_i all the cells occupied by the body of it;
- we save all cartons coordinates in *vertexCoords*.

Second one is ***pushable(box,t,direction)*** which is used to check whether a carton can be pushed toward direction d at time t. Since this predicate considers all directions (which are similar among them), we only show $DOWN$ direction ($d = 0$).

```
predicate pushable(var CARTONS:id , STEPS:t, var MOVES:d) =

     (d == 0 -> (
       exists(i in 0..initialCoords[id,3]-1)(
        room[vertexCoords[id,2, t]-initialCoords[id,3], vertexCoords[id,1,t]+i,t] == 0
       )
       /\

       not exists(i in 0..initialCoords[id,3]-1)(
        room[vertexCoords[id,2, t]+1, vertexCoords[id,1,t]+i,t] != 0
       )
     ))
     /\
     % other directions...
);
```

In order, for each direction d:

- we check whether behind the carton c_{id} (where the sokoban should be) there is *at least* an empty cell;
- we check wheter over the carton (located in x, y) all cells are free. More precisely, if there are no rows r_j such that there exist a cell $room[x+1][j]$ which is occupied.

Third one is ***inertia()*** which preserves positions of unmoved boxes. More precisely, all boxes not moved at time t must have same coords at time t_{i+1}.

```
predicate inertia() =

   (forall(t in STEPS) (
     forall(carton in CARTONS)(
        (carton != boxSelectedAtTime[t] ->  %if a box was not moved
        (vertexCoords[carton,1,t+1] = vertexCoords[carton,1,t] /\  % x untouched
         vertexCoords[carton,2,t+1] = vertexCoords[carton,2,t]))    % y untouched
        )

      /\

      (not exists(carton in CARTONS)(
       not pushable(boxSelectedAtTime[t],t,movesDoneAtTime[t]) /\
       boxSelectedAtTime[t] = carton)
     )
));
```

In order:

- first clause guarantees that all boxes that have not been moved remain in the same position;
- second clause is used to prevent situations in which the are no available moves and the solver tries to do illegal moves.

Fourth predicate is ***move()*** which is used to update coordinates of the moved carton c_i at time t. Again, since it consider all possible movements, some parts are omitted. An interesting case is the null move, for which we do not update coordinates.

```
predicate move() =

 (forall(t in STEPS) (

    % direction D O W N

    ((movesDoneAtTime[t] = 0 /\ pushable(boxSelectedAtTime[t],t,0))->
      (vertexCoords[boxSelectedAtTime[t],1,t+1] = vertexCoords[boxSelectedAtTime[t],1,t] /\
       vertexCoords[boxSelectedAtTime[t],2,t+1] = vertexCoords[boxSelectedAtTime[t],2,t] +1))

    /\

    % other directions ...

    /\

    % none movement

    (movesDoneAtTime[t] = 4 ->
     (vertexCoords[boxSelectedAtTime[t],1,t+1] = vertexCoords[boxSelectedAtTime[t],1,t] /\
      vertexCoords[boxSelectedAtTime[t],2,t+1] = vertexCoords[boxSelectedAtTime[t],2,t]))

));
```

In order:

- first clause checks for selected move $movesDoneAtTime[t]$ if box $boxSelectedAtTime[t]$ is *pushable* toward the selected direction at time t. In this case it considers if the box is pushable toward down.

- if so, it updates coordinates.

Fifth predicate is ***printMatrix()*** which is used to populate the matrix *room* at each steps, using *vertexCoords*. Process is similar to predicate *setInitialMatrix()*. Code is omitted for space issues.
Final predicate is ***finalPosition()*** which assures, as the name suggests, that all boxes at time *maxTime* must be at correct position.

```
predicate finalPosition() =
  forall(box in CARTONS)(
  (vertexCoords[box,1,maxTime] = finalCoords[box,1]) /\
  (vertexCoords[box,2,maxTime] = finalCoords[box,2])
);
```

2.3 Constraints definition

Since we defined each predicate separately, the only constraint is the *conjunction* of all the previous predicates.

```
constraint

  setInitialMatrix() /\ inertia() /\ move() /\ printMatrix() /\ finalPosition()
;
```

2.4 Optimization and Search Strategies

Only a variable has been used, ***nullMoves***, to count number of null moves inside *movesDoneAtTime*.

```
var int: nullMoves = count(move in movesDoneAtTime) (move == 4);
```

Our goal is obviously to minimize the variable *nullMoves*. However, we can leverage the *int_search* to speed up the search for an optimal solution by modifying the search tree traversal strategy.

```
solve
    :: int_search(movesDoneAtTime,first_fail,indomain_min)
    :: int_search(boxSelectedAtTime,first_fail,indomain_random)
    :: restart_linear(10)
maximize nullMoves;
```

All annotations include *first_fail* since we want the computation to stop as soon as possible if the chosen path is wrong. Different is the *constraintchoice* choice: for *movesDoneAtTime* it starts choosing moves from the first value of domain, i.e. 0 (which corresponds to down); this annotation was selected to avoid using *null move*, which is move 4. For *boxSelectedAtTime*, instead, it search values randomically in the defined interval, i.e. it selects random boxes. Finally, *restart_linear* is used to prevent searching over 10 nodes at first step. Solver used is *Gecode 6.3.0*.

2.5 Solutions printing

For completeness we are including also an example of output of the computation, since the author lost couple of hours debugging the print function.

```
initial Matrix:      final Matrix:
 0 0 0 0 0 0          0 0 0 0 0 0       Moves : 0  0  2  2  0  0  2
 0 0 0 0 0 0          0 0 0 0 0 0       Boxes : 2  1  2  2  1  1  1
 0 0 0 0 0 0          0 0 0 0 0 0       Moves count : 7
 0 0 0 1 0 0          0 0 0 0 0 0
 0 0 2 2 0 0          2 2 0 0 0 0
 0 0 2 2 0 0          2 2 1 0 0 0
```

3 ASP Modelling

We will present the ASP model of the problem; it's made of x parts. We will briefly analyze all of them.

3.1 Predicates definition

It follows a list of initial predicates introduced to model problem's data effectively.

```
rows(1..n).
columns(1..m).
number(1..boxNumber).
dimension(1..maxDim).
time(0..maxTime).
direction(u;d;l;r).
boxCoord(Id,L,X,Y).
on(T,Id,X,Y).
vertexPosition(T,Id,X,Y)
```

In order:

- ***rows*** and ***columns*** are used to model the room's size;
- ***number*** identifies every box, since every carton has to be identified uniquely by an id;
- ***dimension*** is used to model the size of each carton;
- ***time*** is used to count steps during the computation;
- ***direction*** is used to indicate which direction the move has to be;
- ***boxCoord*** is used to indicate *initial* coordinates of carton c_{Id};
- ***on*** is used to set location X,Y *occupied* by box c_{Id} at time T;
- ***vertexPosition*** is used to save coordinates of box c_{Id} at time T.

Differences between *on*, *boxCoord* and *vertexPosition* are in their use. First one is meant to save *all* occupied cells by each carton c_{Id} at each step of time T. Second one is used to take values from the input instance; third one is used only to store coordinates of the *vertex* of each box c_{Id} at each step of time T.

3.2 Constraints definition

We used constraints to model each state of the computation; starting point is when time $T = 0$. To model the *initial state* we created the following constraints: first one is used to occupy cells for each box c_{Id} of size L, starting from coordinates X, Y. Second one is used to save vertex coordinates of each box at starting time. When a cell gets occupied we activate the predicate *occupiedCell* hiding details of the carton over it. This practice is useful in situations where we want to check whether a cell is free or not (without caring about which carton stands over it).

```
on(0,Id,X,Y):-boxCoord(Id,L,X1,Y1),X >= X1,X < X1+L,Y >= Y1,Y < Y1+L, columns(X), rows(Y).
vertexPosition(0,Id,X,Y):-boxCoord(Id,L,X,Y).
occupiedCell(T,X,Y):-on(T,_,X,Y).
```

At every state we need that there are no colliding boxes:

```
:- on(T,Id1,X,Y),on(T,Id2,X,Y),Id1!=Id2.
```

When we want to *move* a box we use the same reasoning explained in the MiniZinc part. To simplify the explanation we only consider a movement toward *down* direction. In order:

- we check wheter the box is *pushable*;
- if so, we define the current move as a *validMove*;
- finally, we select *at most* a move from the valid ones for each time step T.

More specifically, we model the *move* toward down using the following constraints:

```
% occupiedRow active indicates that in the cells (Xi,Y) AT LEAST one is occupied.
occupiedRow(T,X,Y,N):-dimension(N),occupiedCell(T,X1,Y),columns(X1),columns(X),rows(Y),
                      X1>=X,X1<X+N,time(T).

% freeRow is true if there is AT LEAST one free cell.
freeRow(T,X,Y,N):-dimension(N),not occupiedCell(T,X1,Y),columns(X1),columns(X),rows(Y),
                      X1>=X,X1<X+N,time(T).

pushable(T,Id,d):-vertexPosition(T,Id,X,Y),not occupiedRow(T,X,Y-1,L),boxCoord(Id,L,_,_),
                  rows(Y-1),columns(X),freeRow(T,X,Y+L,L).

validMoves(T,Id,D):-vertexPosition(T,Id,X,Y),direction(D),pushable(T,Id,D),
                    rows(Y),columns(X).

0{move(T,Id,D):validMoves(T,Id,D)}1:-time(T),T<maxTime.
```

Once the move has been done correctly, we must update positions of moved box while maintain all the others stationary.; we modeled the constraint in the following manner:

```
% if it doesn't move a box, then the box stays in its original position.
on(T+1,Id,X,Y):-on(T,Id,X,Y), not move(T,Id,_),T<maxTime.

vertexPosition(T+1,Id,X,Y):- vertexPosition(T,Id,X,Y), not move(T,Id,_),T<maxTime.

% if it moves a box, it moves both the vertex coordinates and the on's predicate.
on(T+1, Id, XNew, YNew) :- move(T, Id, D), on(T, Id, XOld, YOld),
                           newCoord(XNew, YNew, XOld, YOld, D),T<maxTime.

newCoord(XNew, YNew, XOld, YOld, d) :- XNew = XOld, YNew = YOld-1,columns(XNew),
                                       columns(XOld),rows(YNew),rows(YOld).

vertexPosition(T+1, Id, XNew, YNew) :- move(T, Id, D),vertexPosition(T,Id,XOld,YOld),
                                       newCoord(XNew, YNew, XOld, YOld, D),T<maxTime.
```

Finally, we want to conclude the computation with all boxes standing in the right position, i.e. in the coordinates specified with *deposit*.

```
% GOAL
goal(T,Id) :- vertexPosition(T,Id,X,Y),deposit(Id,X,Y),time(T).
:- goal(T,Id), move(T2,Id,_),time(T2),T2>T,vertexPosition(T,Id,_,_).

%% FINAL STATE
:- not goal(maxTime,Id),vertexPosition(maxTime,Id,_,_).
```

3.3 Optimization

In this model we decided to count the number of *valid* moves using a variable. The reasoning is complementary to the MiniZinc one, since our aim is to *minimize* the variable.

```
requiredMoves(M) :- M = #count{T:move(T,Id,D)}.
#minimize {M:requiredMoves(M)}.
```

3.4 Solutions printing

Always for completeness we are including an example of output of the computation over an instance.

```
move(16,2,d) move(17,1,d) move(18,2,l) move(20,1,d) move(19,2,l)
move(21,1,l) move(23,1,d) finalVertex(2,1,1) finalVertex(1,3,1)
Optimization: 7
```

4 Istances generation

In order to test performance of our models, we need a sufficient number of test cases to run the two solvers; in this analysis we used 30 instances. Thus, we have written a Python script capable of pseudo-randomly generating instances whose difficulty is proportional to the value supplied as input. For space issues the code will be omitted, instead it will be presented only the idea behind it. A brief *pseudocode* of the *generateValues(difficulty)* will be presented.

Algorithm 1 Values generator

```
$values \leftarrow random(difficulty)$
$matrix \leftarrow initializeMatrix(values)$
$initialCoordinates \leftarrow generateBoxes(values)$
$populateMatrix(matrix, initialCoordinates)$
$finalCoordinates \leftarrow generateGoals(values, matrix)$
```

Based on difficulty selected, the program sets parameters of the instance saved in a list *values*, i.e. *number of boxes*, *size of each box*, *size of the room*, ...
Then, it proceeds to generate the initial coordinates of the selected boxes using the following reasoning: *I need to create boxes that are positioned away from the room's borders, with no collisions between them, and preferably with at least one empty cell between any two boxes.* Once vertexes are created, it has to generate *goal* positions. The concept is to begin stacking larger boxes in the bottom-left corner of the room and continue to the right, as shown in figure 2.

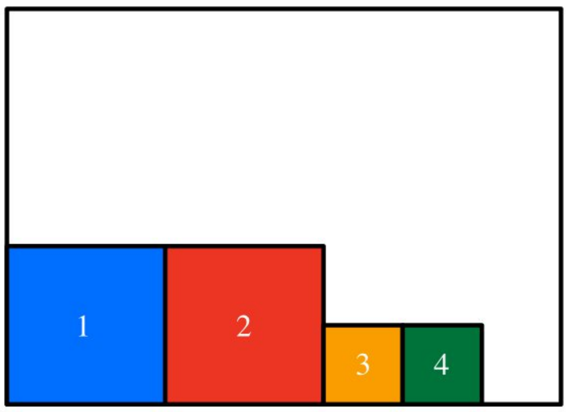

Figure 2: Final cartons disposition.

As already said, we decided to generate 30 instances; respectively:

- 5 *easy* instances: a couple of running time seconds;
- 20 *medium* instances: a couple of minutes;
- 5 *hard* instances: exceeding 5 minutes;

Table 1 summarize all principal characteristics of the generated instances. We will briefly summarize what columns stand for:

- m: represents the rows of the matrix (room);
- n: represents the columns of the matrix (room);
- $BoxNumber$: number of boxes inside the room;
- $MaxDim$: size of the bigger carton inside the room;

Instance number	m	n	BoxNumber	MaxDim
easy 1	3	3	1	2
easy 2	3	3	1	2
easy 3	3	3	1	2
easy 4	4	5	1	3
easy 5	3	3	1	2
medium 1	5	5	2	3
medium 2	5	5	2	3
medium 3	5	5	1	3
medium 3	5	5	1	3
medium 4	5	5	2	3
medium 5	5	4	1	3
medium 6	4	5	2	3
medium 7	4	3	2	2
medium 8	4	4	1	3
medium 9	5	4	1	3
medium 10	4	4	2	3
medium 11	4	4	1	3
medium 12	4	4	1	3
medium 13	5	4	1	3
medium 14	4	5	1	3
medium 15	5	5	1	3
medium 16	4	5	1	3
medium 17	4	5	1	3
medium 18	4	4	1	3
medium 19	4	5	2	3
medium 20	5	5	1	3
hard 1	8	8	1	4
hard 2	8	8	3	4
hard 3	8	8	2	4
hard 4	8	8	3	4
hard 5	7	7	3	4

Table 1: Instances properties

5 Performance analysis

We will present benchmarks acquired using both models on all instances. Table 2 contains all results obtained and times measured. In order:

- $best(moves)$: represent the minor number of moves that model's found;
- $time\_found(s)$: is the time, in seconds, in which the model found the solution in the *best* column;
- $total\_time(s)$ is the total time used by the model to explore the computation tree.

In cases where e is written it means that computation exceed 5 minutes (even if the correct solution was found). As we can see, *ASP model* perfomed better than MZN one almost for every easy and medium instance; the results are opposite in hard instance cases, since in some computations *ASP model* was not even able to find the right minimum. This is probably because the heuristics used perform better with hard instances. A visual representation is given in plots 3, 4; both use a logarithmic scale on the y-axis since the times interval is slightly big. The first chart represents times taken by the two models to compute the optimal solution, while the second one represents times of total computations of the two models. The upperbound represents e, which stands for 300s.

	ASP			**MZN**		
	best (moves)	time_found (s)	total time (s)	best (moves)	time_found (s)	total time (s)
e1	2	0.024	0.024	2	1.802	1.802
e2	2	0.023	0.023	2	1.390	2.100
e3	2	0.024	0.024	2	0.190	1.756
e4	1	0.022	0.022	1	0.320	0.320
e5	2	0.023	0.023	2	1.339	1.339
m1	8	0.870	e	8	0.910	e
m2	5	0.270	e	5	0.750	e
m3	3	0.430	0.486	3	0.599	e
m4	5	0.480	158.699	5	6.130	e
m5	3	0.350	0.397	3	0.810	e
m6	4	11.840	11.899	4	1.68	e
m7	4	1.970	1.995	4	0.191	e
m8	3	0.250	0.285	3	0.830	e
m9	2	0.058	0.060	2	0.660	4.168
m10	4	8.41	9.465	4	0.890	e
m11	4	2.18	2.215	2	0.601	e
m12	3	0.250	0.289	3	0.610	e
m13	5	13.75	13.79	5	0.760	e
m14	3	0.370	0.411	3	0.680	e
m15	4	3.87	3.914	4	0.760	e
m16	4	3.10	3.147	4	0.630	e
m17	2	0.054	0.060	2	0.960	3.631
m18	3	0.250	0.283	3	0.566	e
m19	5	17.00	207.847	5	35.290	e
m20	3	0.440	0.488	3	0.680	e
h1	6	14.15	e	6	2.810	e
h2	48	112.10	e	16	11.470	e
h3	12	123.65	e	12	108.150	e
h4	37	267.31	e	19	214.000	e
h5	9	146.00	e	9	14.790	e

Table 2: Benchmarks

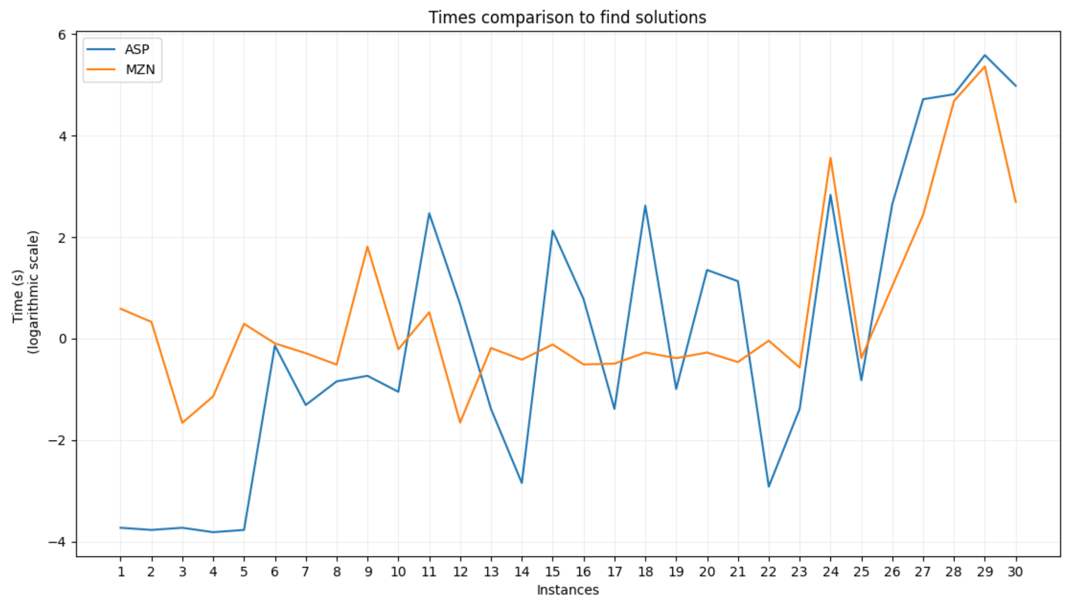

Figure 3: Benchmarks of solutions time, in logarithmic scale.

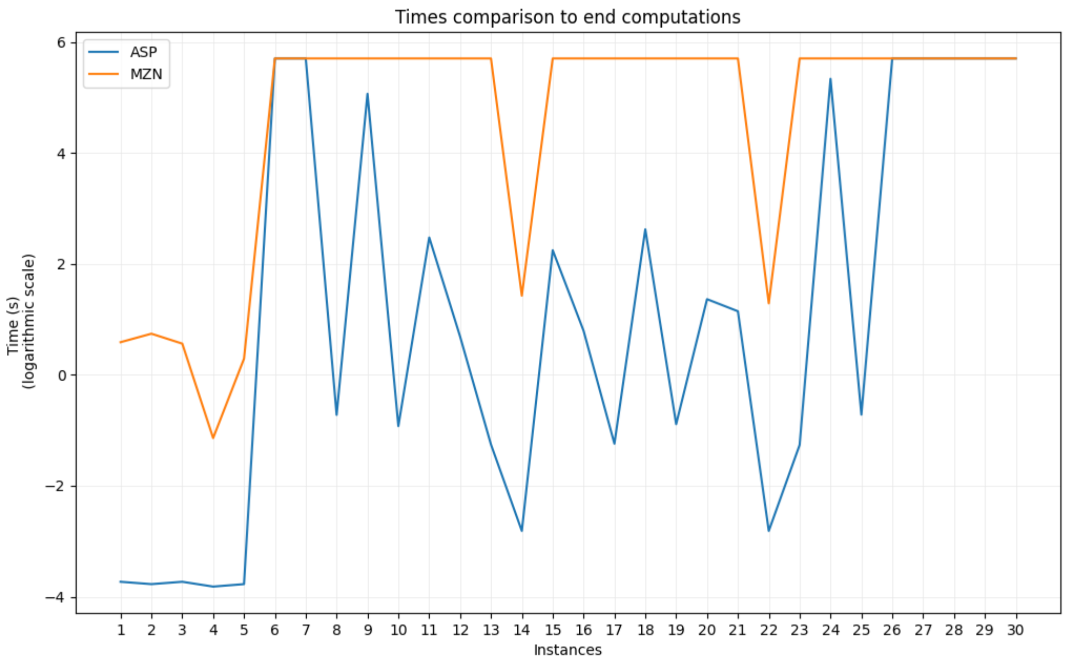

Figure 4: Benchmarks of computations time, in logarithmic scale.

References

[1] Andreas Junghanns and Jonathan Schaeffer. Sokoban: A challenging single-agent search problem. In *International Joint Conference on Artificial Intelligence*, 1997.

[2] neng-fa Zhou and Agostino Dovier. A tabled prolog program for solving sokoban. *Fundamenta Informaticae*, 124:561–575, 01 2013.