Solving a game of domino with MiniZinc and ASP

The problem is as follows:

> Problem definition
>
> Given a multiset of stones, each made of two labelled sides, try to place as many as possible of them on an $n \times n$ grid while respecting the following rules:
>
> - Stones may not overlap, neither partially nor fully
> - Stones may not leave the board, neither partially nor fully
> - Each stone shall be placed so that at least one of its sides is adjacent to a side of the previous stone and with matching label
> - Each stone may only be placed once
>
>   Stones are labelled with numbers from 0 to 6 included

A simple python program has been written to automatically generate instances and benchmark various models with them while recording solutions and statistics. Models, python program and benchmarks can all be found in this repository

Code snippets of MiniZinc and ASP will be shown as screenshots rather than code blocks due to the lack of syntax highlighting

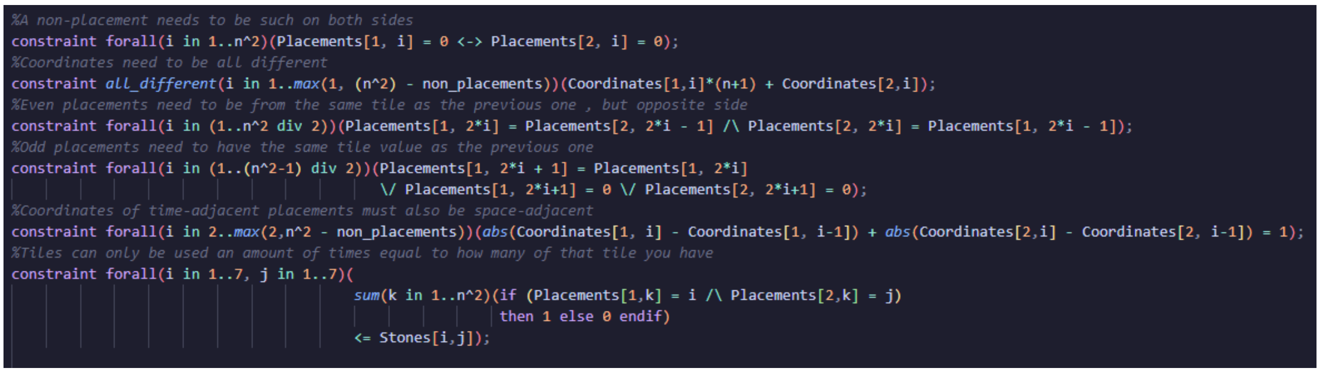
MiniZinc Program

The MiniZinc model expects the multiset of stones as an $n \times n$ matrix S where element $S[a, b]$ indicates the amount of available stones with label (a, b). Such an input format has been chosen to inherently combat identical-stones symmetries (i.e. situations where the solver will try to swap two identical stones when trying to find a better solution)

The solution is calculated as two $2 \times n^2$ matrixes P and C (`Placements` and `Coordinates` respectively in the program). $P[1, n]$ indicates the stone label used at time n while $P[2, n]$ indicates the label on the "other side" of that stone. Similarly $C[1, n]$ and $C[2, n]$ indicate the two coordinates of the n-th placement

Number labels are increased by 1 while within the program so that 0 can be used as a special value that indicates missing placements

The constraints that define the validity of a solution candidate are then as follows:



```
%A non-placement needs to be such on both sides
constraint forall(i in 1..n^2)(Placements[1, i] = 0 <-> Placements[2, i] = 0);
%Coordinates need to be all different
constraint all_different(i in 1..max(1, (n^2) - non_placements))(Coordinates[1,i]*(n+1) + Coordinates[2,i]);
%Even placements need to be from the same tile as the previous one , but opposite side
constraint forall(i in (1..n^2 div 2))(Placements[1, 2*i] = Placements[2, 2*i - 1] /\ Placements[2, 2*i] = Placements[1, 2*i - 1]);
%Odd placements need to have the same tile value as the previous one
constraint forall(i in (1..(n^2-1) div 2))(Placements[1, 2*i + 1] = Placements[1, 2*i]
                                           \/ Placements[1, 2*i+1] = 0 \/ Placements[2, 2*i+1] = 0);
%Coordinates of time-adjacent placements must also be space-adjacent
constraint forall(i in 2..max(2,n^2 - non_placements))(abs(Coordinates[1, i] - Coordinates[1, i-1]) + abs(Coordinates[2,i] - Coordinates[2, i-1]) = 1);
%Tiles can only be used an amount of times equal to how many of that tile you have
constraint forall(i in 1..7, j in 1..7)(
                                        sum(k in 1..n^2)(if (Placements[1,k] = i /\ Placements[2,k] = j)
                                                         then 1 else 0 endif)
                                        <= Stones[i,j]);
```

Please note how each placement describes only "half of a stone" rather than the full stone. For example a solution for a 4×4 grid that placed 8 stones will have 16 "placements". It is for this reason that even and odd numbered placements are treated differently: the former are always the "other side" to the previous placement rather than a new stone

Symmetry breaking

Several additional rules were then added to break symmetries, reducing the space of possible solutions

The most impactful and most numerous of those are related to a unique trait of the problem: the placement of the tiles on the grid and their order are two entirely separate problems and the former has a trivial solution

That is because every stone placement only cares about the previous placements and no other stone. This allows us to force the program to place the stones in a "snake-like" orderly pattern as such:

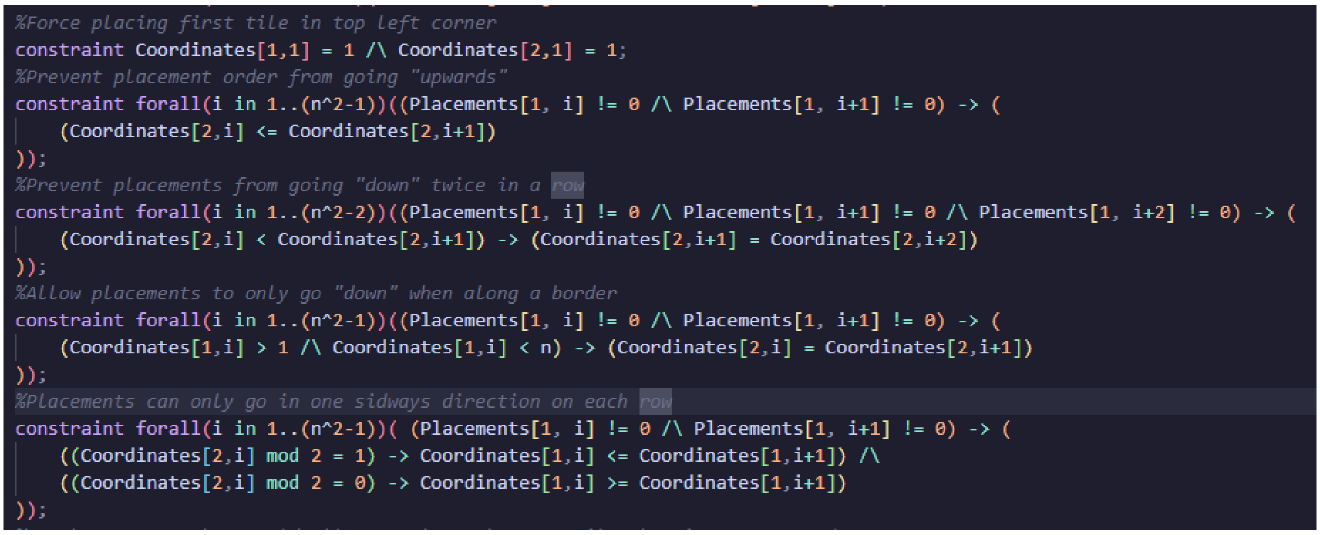
```
0 0 0 1 1
| | | | |
2 4 1 3 3


2 4 1 3 3
| | | | |
5 5 1 1 1
```

```
5-5 1-1 #
```

Where the top left corner is always the starting point (`#` represents an empty space and the two sides of a single stone are connected by a line)

This is achieved by using the following constraints

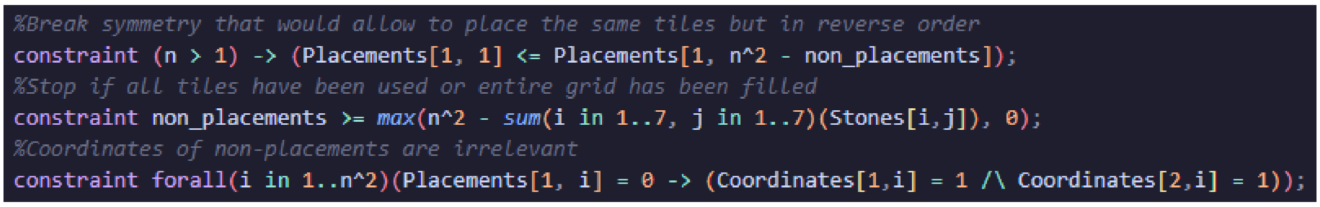


```
%Force placing first tile in top left corner
constraint Coordinates[1,1] = 1 /\ Coordinates[2,1] = 1;
%Prevent placement order from going "upwards"
constraint forall(i in 1..(n^2-1))((Placements[1, i] != 0 /\ Placements[1, i+1] != 0) -> (
    (Coordinates[2,i] <= Coordinates[2,i+1])
));
%Prevent placements from going "down" twice in a row
constraint forall(i in 1..(n^2-2))((Placements[1, i] != 0 /\ Placements[1, i+1] != 0 /\ Placements[1, i+2] != 0) -> (
    (Coordinates[2,i] < Coordinates[2,i+1]) -> (Coordinates[2,i+1] = Coordinates[2,i+2])
));
%Allow placements to only go "down" when along a border
constraint forall(i in 1..(n^2-1))((Placements[1, i] != 0 /\ Placements[1, i+1] != 0) -> (
    (Coordinates[1,i] > 1 /\ Coordinates[1,i] < n) -> (Coordinates[2,i] = Coordinates[2,i+1])
));
%Placements can only go in one sidways direction on each row
constraint forall(i in 1..(n^2-1))( (Placements[1, i] != 0 /\ Placements[1, i+1] != 0) -> (
    ((Coordinates[2,i] mod 2 = 1) -> Coordinates[1,i] <= Coordinates[1,i+1]) /\
    ((Coordinates[2,i] mod 2 = 0) -> Coordinates[1,i] >= Coordinates[1,i+1])
));
```

The performance impact of such constraints is massive, going from taking over 20 seconds to solve an 11×11 instance to just under 2 seconds

---

Some other constraints have been also added to improve performance



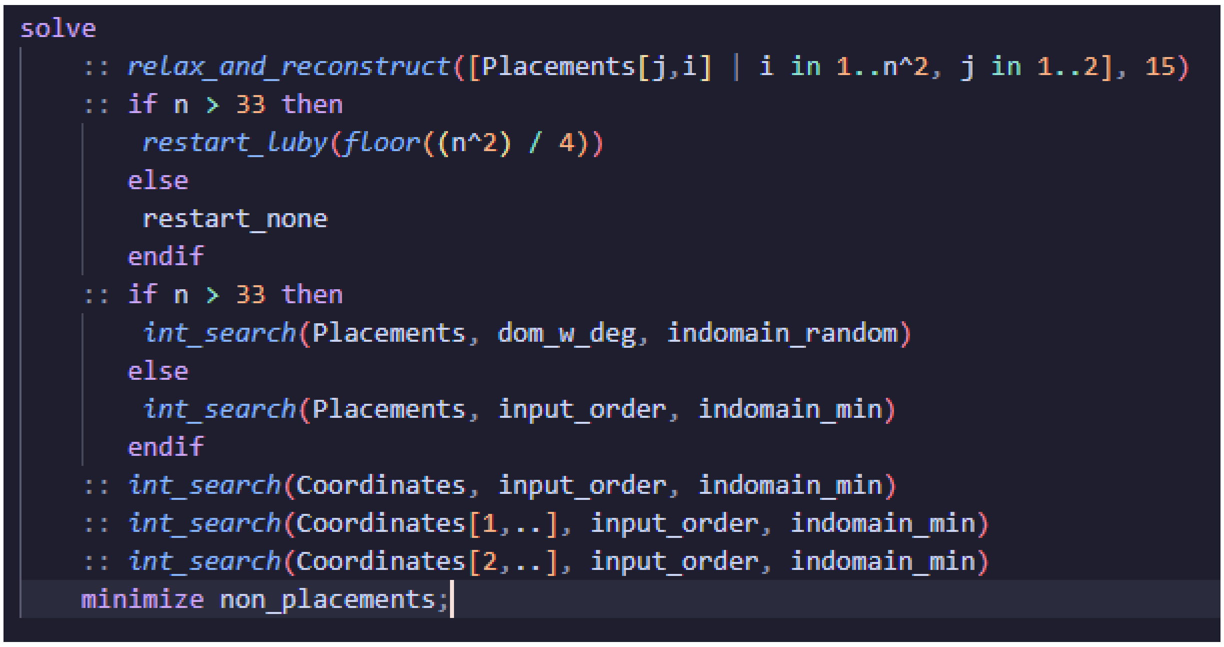
```
%Break symmetry that would allow to place the same tiles but in reverse order
constraint (n > 1) -> (Placements[1, 1] <= Placements[1, n^2 - non_placements]);
%Stop if all tiles have been used or entire grid has been filled
constraint non_placements >= max(n^2 - sum(i in 1..7, j in 1..7)(Stones[i,j]), 0);
%Coordinates of non-placements are irrelevant
constraint forall(i in 1..n^2)(Placements[1, i] = 0 -> (Coordinates[1,i] = 1 /\ Coordinates[2,i] = 1));
```

Iterative improvements through benchmarking

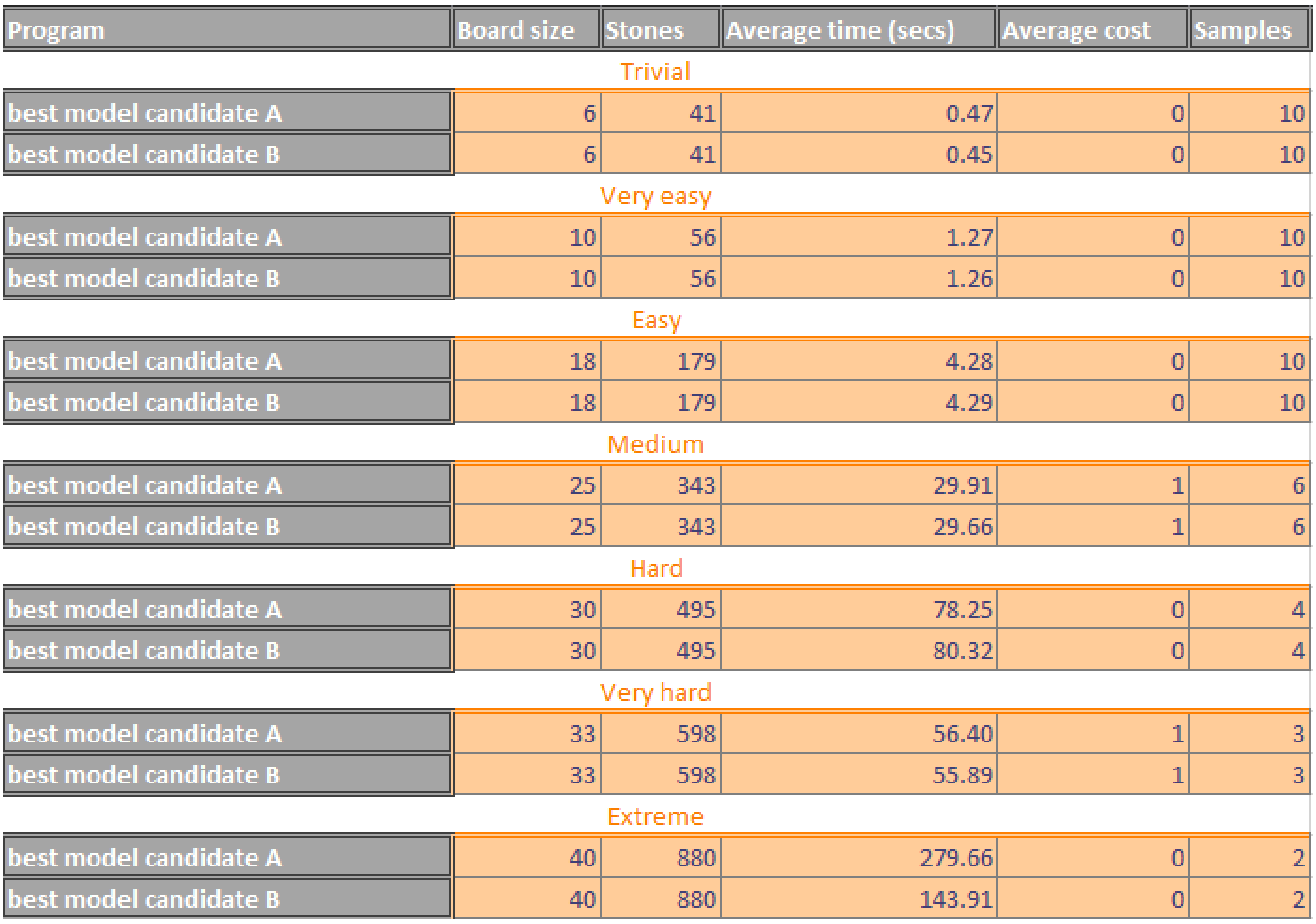
By analyzing the results of multiple benchmarks, several different performance enhancing features have been added to the program while iterating upon it. All such benchmarks can be found in the `benchmarks` folder, where the statistics and results are stored in a labelled CSV format and the solution to each specific instance is stored in a separate file in a human-readable format

To save some time each program being tested had a total maximum amount of time allowed for each benchmark batch. Once it ran out of such time it wasn't allowed to even attempt the harder instances in the batch. This maximum time has been set to be very generous however so that the benchmarks could still be useful and exhaustive for the better and more interesting programs

The first few tests where trying to check the effectiveness of the additional constraints explained above. After that various MiniZinc heuristics were tested. Finally several benchmarks have been ran to find the best restart strategy and the results are as follows:



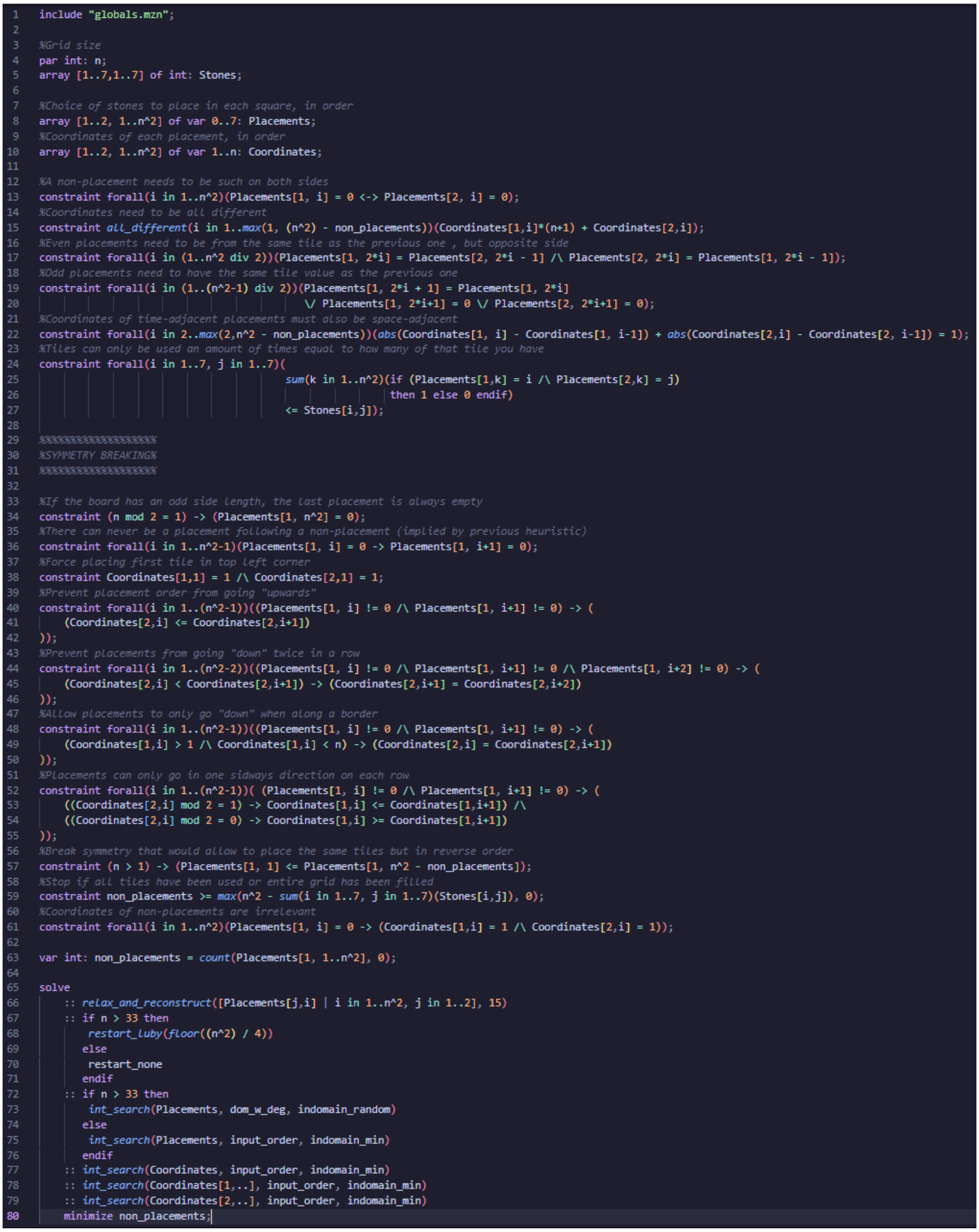
```
solve
    :: relax_and_reconstruct([Placements[j,i] | i in 1..n^2, j in 1..2], 15)
    :: if n > 33 then
        restart_luby(floor((n^2) / 4))
       else
        restart_none
       endif
    :: if n > 33 then
        int_search(Placements, dom_w_deg, indomain_random)
       else
        int_search(Placements, input_order, indomain_min)
       endif
    :: int_search(Coordinates, input_order, indomain_min)
    :: int_search(Coordinates[1,..], input_order, indomain_min)
    :: int_search(Coordinates[2,..], input_order, indomain_min)
    minimize non_placements;
```



Program	Board size	Stones	Average time (secs)	Average cost	Samples
		Trivial			
best model candidate A	6	41	0.47	0	10
best model candidate B	6	41	0.45	0	10
		Very easy			
best model candidate A	10	56	1.27	0	10
best model candidate B	10	56	1.26	0	10
		Easy			
best model candidate A	18	179	4.28	0	10
best model candidate B	18	179	4.29	0	10
		Medium			
best model candidate A	25	343	29.91	1	6
best model candidate B	25	343	29.66	1	6
		Hard			
best model candidate A	30	495	78.25	0	4
best model candidate B	30	495	80.32	0	4
		Very hard			
best model candidate A	33	598	56.40	1	3
best model candidate B	33	598	55.89	1	3
		Extreme			
best model candidate A	40	880	279.66	0	2
best model candidate B	40	880	143.91	0	2

Unfortunately any instance bigger than 40×40 would result in memory issues on the benchmarking machine and thus couldn't be tested

The best program in full is thus as follows:



```
include "globals.mzn";

%Grid size
par int: n;
array [1..7,1..7] of int: Stones;

%Choice of stones to place in each square, in order
array [1..2, 1..n^2] of var 0..7: Placements;
%Coordinates of each placement, in order
array [1..2, 1..n^2] of var 1..n: Coordinates;

%A non-placement needs to be such on both sides
constraint forall(i in 1..n^2)(Placements[1, i] = 0 <-> Placements[2, i] = 0);
%Coordinates need to be all different
constraint all_different(i in 1..max(1, (n^2) - non_placements))(Coordinates[1,i]*(n+1) + Coordinates[2,i]);
%Even placements need to be from the same tile as the previous one , but opposite side
constraint forall(i in (1..n^2 div 2))(Placements[1, 2*i] = Placements[2, 2*i - 1] /\ Placements[2, 2*i] = Placements[1, 2*i - 1]);
%Odd placements need to have the same tile value as the previous one
constraint forall(i in (1..(n^2-1) div 2))(Placements[1, 2*i + 1] = Placements[1, 2*i]
                                           \/ Placements[1, 2*i+1] = 0 \/ Placements[2, 2*i+1] = 0);
%Coordinates of time-adjacent placements must also be space-adjacent
constraint forall(i in 2..max(2,n^2 - non_placements))(abs(Coordinates[1, i] - Coordinates[1, i-1]) + abs(Coordinates[2,i] - Coordinates[2, i-1]) = 1);
%Tiles can only be used an amount of times equal to how many of that tile you have
constraint forall(i in 1..7, j in 1..7)(
                                        sum(k in 1..n^2)(if (Placements[1,k] = i /\ Placements[2,k] = j)
                                                         then 1 else 0 endif)
                                        <= Stones[i,j]);

%%%%%%%%%%%%%%%%%%%
%SYMMETRY BREAKING%
%%%%%%%%%%%%%%%%%%%

%If the board has an odd side length, the last placement is always empty
constraint (n mod 2 = 1) -> (Placements[1, n^2] = 0);
%There can never be a placement following a non-placement (implied by previous heuristic)
constraint forall(i in 1..n^2-1)(Placements[1, i] = 0 -> Placements[1, i+1] = 0);
%Force placing first tile in top left corner
constraint Coordinates[1,1] = 1 /\ Coordinates[2,1] = 1;
%Prevent placement order from going "upwards"
constraint forall(i in 1..(n^2-1))((Placements[1, i] != 0 /\ Placements[1, i+1] != 0) -> (
    (Coordinates[2,i] <= Coordinates[2,i+1])
));
%Prevent placements from going "down" twice in a row
constraint forall(i in 1..(n^2-2))((Placements[1, i] != 0 /\ Placements[1, i+1] != 0 /\ Placements[1, i+2] != 0) -> (
    (Coordinates[2,i] < Coordinates[2,i+1]) -> (Coordinates[2,i+1] = Coordinates[2,i+2])
));
%Allow placements to only go "down" when along a border
constraint forall(i in 1..(n^2-1))((Placements[1, i] != 0 /\ Placements[1, i+1] != 0) -> (
    (Coordinates[1,i] > 1 /\ Coordinates[1,i] < n) -> (Coordinates[2,i] = Coordinates[2,i+1])
));
%Placements can only go in one sidways direction on each row
constraint forall(i in 1..(n^2-1))( (Placements[1, i] != 0 /\ Placements[1, i+1] != 0) -> (
    ((Coordinates[2,i] mod 2 = 1) -> Coordinates[1,i] <= Coordinates[1,i+1]) /\
    ((Coordinates[2,i] mod 2 = 0) -> Coordinates[1,i] >= Coordinates[1,i+1])
));
%Break symmetry that would allow to place the same tiles but in reverse order
constraint (n > 1) -> (Placements[1, 1] <= Placements[1, n^2 - non_placements]);
%Stop if all tiles have been used or entire grid has been filled
constraint non_placements >= max(n^2 - sum(i in 1..7, j in 1..7)(Stones[i,j]), 0);
%Coordinates of non-placements are irrelevant
constraint forall(i in 1..n^2)(Placements[1, i] = 0 -> (Coordinates[1,i] = 1 /\ Coordinates[2,i] = 1));

var int: non_placements = count(Placements[1, 1..n^2], 0);

solve
    :: relax_and_reconstruct([Placements[j,i] | i in 1..n^2, j in 1..2], 15)
    :: if n > 33 then
         restart_luby(floor((n^2) / 4))
        else
         restart_none
        endif
    :: if n > 33 then
         int_search(Placements, dom_w_deg, indomain_random)
        else
         int_search(Placements, input_order, indomain_min)
        endif
    :: int_search(Coordinates, input_order, indomain_min)
    :: int_search(Coordinates[1,..], input_order, indomain_min)
    :: int_search(Coordinates[2,..], input_order, indomain_min)
    minimize non_placements;
```

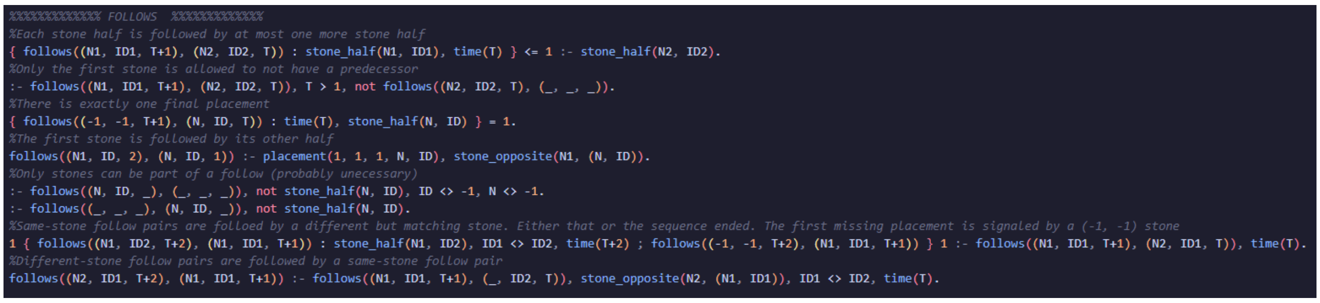
ASP Program

The ASP Program has been developed by first trying to "port over" the same ideas used in MiniZinc but that route quickly proved itself to be wrong and inefficient

A novel approach was thus needed and so the ASP model ended up being based on `follow` predicates. In this program each stone is given as a `stone((N1, N2), ID)` fact where `(N1, N2)` are the numbers on the stone and `ID` is a unique identifier for it so that multiple identical stones could be provided

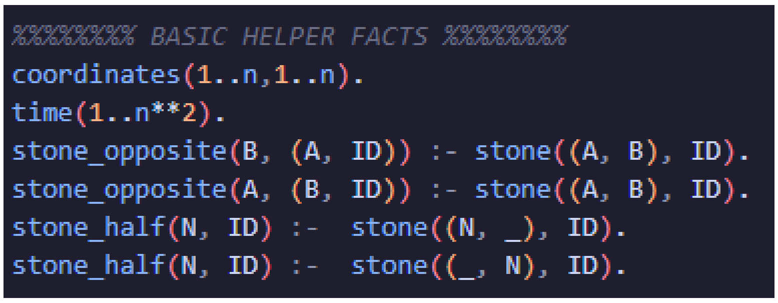
A `follow((N1, ID1, T1), (N2, ID2, T2))` predicate thus indicates that the number `N2` from the stone identified by `ID2` has been placed at time step `T2` and is followed by the number `N1` from the stone identified by `ID1` at time step `T1`. Similarly to before we work with single numbers and single cells rather than entire stones at once. The reason for the inclusion of a time variable in the `follows` predicate will be explained later

Such predicates are handled by the following rules:



```
%%%%%%%%%%%%% FOLLOWS  %%%%%%%%%%%%%
%Each stone half is followed by at most one more stone half
{ follows((N1, ID1, T+1), (N2, ID2, T)) : stone_half(N1, ID1), time(T) } <= 1 :- stone_half(N2, ID2).
%Only the first stone is allowed to not have a predecessor
:- follows((N1, ID1, T+1), (N2, ID2, T)), T > 1, not follows((N2, ID2, T), (_, _, _)).
%There is exactly one final placement
{ follows((-1, -1, T+1), (N, ID, T)) : time(T), stone_half(N, ID) } = 1.
%The first stone is followed by its other half
follows((N1, ID, 2), (N, ID, 1)) :- placement(1, 1, 1, N, ID), stone_opposite(N1, (N, ID)).
%Only stones can be part of a follow (probably unecessary)
:- follows((N, ID, _), (_, _, _)), not stone_half(N, ID), ID <> -1, N <> -1.
:- follows((_, _, _), (N, ID, _)), not stone_half(N, ID).
%Same-stone follow pairs are folloed by a different but matching stone. Either that or the sequence ended. The first missing placement is signaled by a (-1, -1) stone
1 { follows((N1, ID2, T+2), (N1, ID1, T+1)) : stone_half(N1, ID2), ID1 <> ID2, time(T+2) ; follows((-1, -1, T+2), (N1, ID1, T+1)) } 1 :- follows((N1, ID1, T+1), (N2, ID1, T)), time(T).
%Different-stone follow pairs are followed by a same-stone follow pair
follows((N2, ID1, T+2), (N1, ID1, T+1)) :- follows((N1, ID1, T+1), (_, ID2, T)), stone_opposite(N2, (N1, ID1)), ID1 <> ID2, time(T).
```

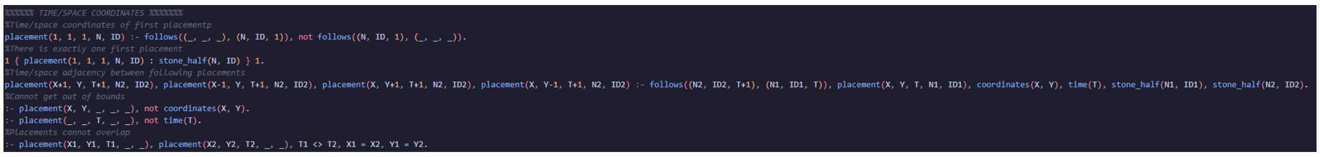
It is worth noting that the triple `(-1, -1, T)` in the left side of a `follow` indicates that the last placement has been made at time `T-1` and such placement thus doesn't have any following it. Most of those rules are self-explanatory but they often use some helper predicates defined as such:



```
%%%%%%%% BASIC HELPER FACTS %%%%%%%%
coordinates(1..n,1..n).
time(1..n**2).
stone_opposite(B, (A, ID)) :- stone((A, B), ID).
stone_opposite(A, (B, ID)) :- stone((A, B), ID).
stone_half(N, ID) :-  stone((N, _), ID).
stone_half(N, ID) :-  stone((_, N), ID).
```

The last two rules for the `follow` predicate are the most relevant ones: the former defines the rules of how a stone can follow another one while the latter ensures that every half-stone placed is followed by its other half. Rather than checking whether the time is odd or even like in MiniZinc these two rules alternate each other by checking the "shape" of the `follow` that came before the one being currently analyzed

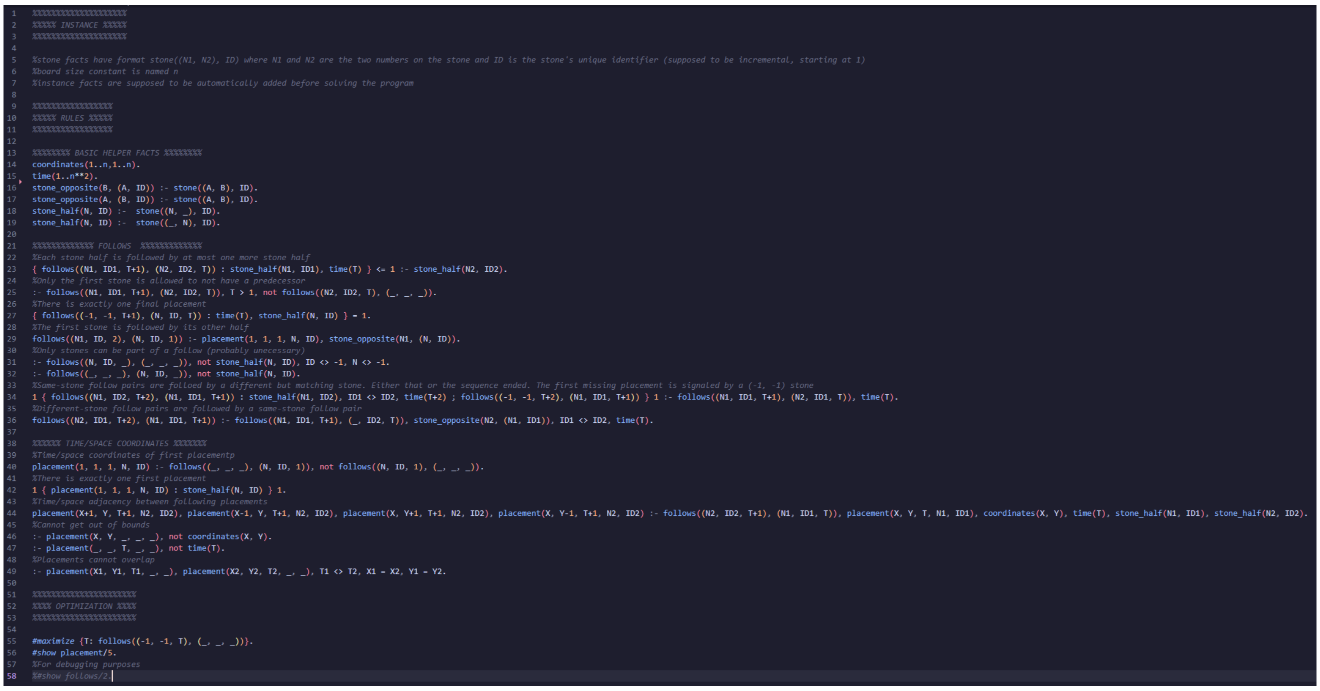
Both to decide the coordinates of each placement and to conform to a certain expected output format all the `follow` predicates are then used to infer `placement(X, Y, T, N, ID)` predicates which indicate that number `N` from the stone identified by `ID` has been placed at coordinates `X, Y` at time step `T` . This is achieved using the following rules:



```
%%%%%% TIME/SPACE COORDINATES %%%%%%%
%Time/space coordinates of first placementp
placement(1, 1, 1, N, ID) :- follows((_, _, _), (N, ID, 1)), not follows((N, ID, 1), (_, _, _)).
%There is exactly one first placement
1 { placement(1, 1, 1, N, ID) : stone_half(N, ID) } 1.
%Time/space adjacency between following placements
placement(X+1, Y, T+1, N2, ID2), placement(X-1, Y, T+1, N2, ID2), placement(X, Y+1, T+1, N2, ID2), placement(X, Y-1, T+1, N2, ID2) :- follows((N2, ID2, T+1), (N1, ID1, T)), placement(X, Y, T, N1, ID1), coordinates(X, Y), time(T), stone_half(N1, ID1), stone_half(N2, ID2).
%Cannot get out of bounds
:- placement(X, Y, _, _, _), not coordinates(X, Y).
:- placement(_, _, T, _, _), not time(T).
%Placements cannot overlap
:- placement(X1, Y1, T1, _, _), placement(X2, Y2, T2, _, _), T1 <> T2, X1 = X2, Y1 = Y2.
```

`follow` utilizes a time step variable `T` to distinguish the left and the right triple in every possible scenario so that the third `placement` rule cannot recursively "place" the same `follow` predicate over and over. Without the time step variable or a similar additional variable a predicate such as `follow((2,4), (2,4))` would result in the stone `stone((2,2), 4)` being placed over and over again over the entire grid

And finally the time of the final placement is maximized, resulting in the following complete program:



```
%%%%%%%%%%%%%%%%%%%%
%%%%% INSTANCE %%%%%
%%%%%%%%%%%%%%%%%%%%

%stone facts have format stone((N1, N2), ID) where N1 and N2 are the two numbers on the stone and ID is the stone's unique identifier (supposed to be incremental, starting at 1)
%board size constant is named n
%instance facts are supposed to be automatically added before solving the program

%%%%%%%%%%%%%%%%%
%%%%% RULES %%%%%
%%%%%%%%%%%%%%%%%

%%%%%%%% BASIC HELPER FACTS %%%%%%%%
coordinates(1..n,1..n).
time(1..n**2).
stone_opposite(B, (A, ID)) :- stone((A, B), ID).
stone_opposite(A, (B, ID)) :- stone((A, B), ID).
stone_half(N, ID) :-  stone((N, _), ID).
stone_half(N, ID) :-  stone((_, N), ID).

%%%%%%%%%%%%% FOLLOWS  %%%%%%%%%%%%%
%Each stone half is followed by at most one more stone half
{ follows((N1, ID1, T+1), (N2, ID2, T)) : stone_half(N1, ID1), time(T) } <= 1 :- stone_half(N2, ID2).
%Only the first stone is allowed to not have a predecessor
:- follows((N1, ID1, T+1), (N2, ID2, T)), T > 1, not follows((N2, ID2, T), (_, _, _)).
%There is exactly one final placement
{ follows((-1, -1, T+1), (N, ID, T)) : time(T), stone_half(N, ID) } = 1.
%The first stone is followed by its other half
follows((N1, ID, 2), (N, ID, 1)) :- placement(1, 1, 1, N, ID), stone_opposite(N1, (N, ID)).
%Only stones can be part of a follow (probably unecessary)
:- follows((N, ID, _), (_, _, _)), not stone_half(N, ID), ID <> -1, N <> -1.
:- follows((_, _, _), (N, ID, _)), not stone_half(N, ID).
%Same-stone follow pairs are folloed by a different but matching stone. Either that or the sequence ended. The first missing placement is signaled by a (-1, -1) stone
1 { follows((N1, ID2, T+2), (N1, ID1, T+1)) : stone_half(N1, ID2), ID1 <> ID2, time(T+2) ; follows((-1, -1, T+2), (N1, ID1, T+1)) } 1 :- follows((N1, ID1, T+1), (N2, ID1, T)), time(T).
%Different-stone follow pairs are followed by a same-stone follow pair
follows((N2, ID1, T+2), (N1, ID1, T+1)) :- follows((N1, ID1, T+1), (_, ID2, T)), stone_opposite(N2, (N1, ID1)), ID1 <> ID2, time(T).

%%%%%% TIME/SPACE COORDINATES %%%%%%%
%Time/space coordinates of first placementp
placement(1, 1, 1, N, ID) :- follows((_, _, _), (N, ID, 1)), not follows((N, ID, 1), (_, _, _)).
%There is exactly one first placement
1 { placement(1, 1, 1, N, ID) : stone_half(N, ID) } 1.
%Time/space adjacency between following placements
placement(X+1, Y, T+1, N2, ID2), placement(X-1, Y, T+1, N2, ID2), placement(X, Y+1, T+1, N2, ID2), placement(X, Y-1, T+1, N2, ID2) :- follows((N2, ID2, T+1), (N1, ID1, T)), placement(X, Y, T, N1, ID1), coordinates(X, Y), time(T), stone_half(N1, ID1), stone_half(N2, ID2).
%Cannot get out of bounds
:- placement(X, Y, _, _, _), not coordinates(X, Y).
:- placement(_, _, T, _, _), not time(T).
%Placements cannot overlap
:- placement(X1, Y1, T1, _, _), placement(X2, Y2, T2, _, _), T1 <> T2, X1 = X2, Y1 = Y2.

%%%%%%%%%%%%%%%%%%%%%%
%%%% OPTIMIZATION %%%%
%%%%%%%%%%%%%%%%%%%%%%

#maximize {T: follows((-1, -1, T), (_, _, _))}.
#show placement/5.
%For debugging purposes
%#show follows/2.
```

This program, however, is considerably slower than those defined using MiniZinc, as shown by the following benchmarks:

Program	Board size	Stones	Average time (secs)	Average cost	Samples
		Easy			
Clingo follows-based model	4	13	0.20	0	20
		Medium			
Clingo follows-based model	5	15	57.33	1.2	10
		Hard			
Clingo follows-based model	7	27	395.26	5	1