

The Room Walk:
a constraint optimization problem

`federico.serafini@studenti.unipr.it`

30/08/2022

Contents

1	The problem	2
2	Modeling	2
2.1	Parameters	2
2.2	Variables	3
2.3	Constraints	3
2.3.1	Static constraints	3
2.3.2	Dynamic constraints	4
2.3.3	Length and turns	4
2.4	Cost function minimization	5
2.5	Additional constraints to help the search	6
3	MiniZinc	7
3.1	Search strategy	8
4	Results	9

1 The problem

Let us consider a $n \times n$ shaped room, with access door I located at cell $(1, 1)$ and exit door O located at cell (n, n) (see picture, with $n = 6$). The room can contain $1 \times k$ or $k \times 1$ walls. The allowed moves are either horizontal or vertical. The problem is to identify a (the) minimal length path from I to O . As tie breaking for length, less turns in the path are preferred. If necessary, you can solve the corresponding decision problem based on path length.

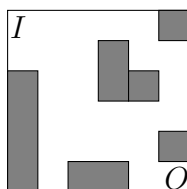


Figure 1: 6×6 room

2 Modeling

The room and the path are clearly the key elements and the problem description is suggesting us to represent the first of them as a matrix. Instead of representing both the room and the path using a single and more complex matrix, two different matrix are used and the relations between them are built using constraints.

2.1 Parameters

- Room size $n \in \mathbb{N}$.
- Room R is a $n \times n$ binary matrix, where $\forall i, j \in [1 \dots n]$:

$$R[i, j] = \begin{cases} 1, & \text{if there is a wall;} \\ 0, & \text{otherwise.} \end{cases}$$

R is randomly generated using a C program and all of its values are set and defined (no decision variables in R , it is an input parameter of our model).

2.2 Variables

- Path P is a $n + 2 \times n + 2$ binary matrix, where $\forall i, j \in [1 \dots n + 2]$:

$$P[i, j] = \begin{cases} 1, & \text{if it is a step of the path;} \\ 0, & \text{otherwise.} \end{cases}$$

P contains the decision variables of our model and its increased size wrt R is due to the zero-padding constraints, needed to simplify a lot the neighborhood exploration (see later).

2.3 Constraints

2.3.1 Static constraints

- Zero padding (first row, first column, last row, last column):

$$\begin{aligned} & \forall j \in [1 \dots n + 2] : P[1, j] = 0 \\ & \wedge \\ & \forall i \in [1 \dots n + 2] : P[i, 1] = 0 \\ & \wedge \\ & \forall j \in [1 \dots n + 2] : P[n + 2, j] = 0 \\ & \wedge \\ & \forall i \in [1 \dots n + 2] : P[i, n + 2] = 0. \end{aligned}$$

- Enter and exit cells (both must be part of the path):

$$P[2, 2] = 1 \wedge P[n + 1, n + 1] = 1.$$

- Avoid walls (if there is a wall then the path can not step over it):

$$\forall i, j \in [1 \dots n] : R[i, j] > 0 \implies P[i, j] = 0.$$

Taking example room in figure (1) as input matrix R , the application of all the static constraints leads the matrix P to look like the following:

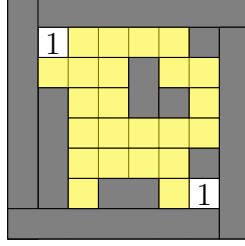


Figure 2: Matrix P after static constraints

Where:

$$P[i, j] = \begin{cases} 0, & \text{if cell is gray;} \\ 0 \vee 1, & \text{if cell is yellow (free variable);} \\ 1, & \text{otherwise.} \end{cases}$$

2.3.2 Dynamic constraints

Now it is time to insert the constraints needed to connect the enter point with the exit point in order to create a path between them.

- Neighborhood for enter and exit cells (exactly one neighbor each):

$$P[2, 3] + P[3, 2] = 1 \wedge P[n + 1, n] + P[n, n + 1] = 1;$$

- Neighborhood for all the other cells in the path (exactly two neighbors each):

$$\begin{aligned} \forall i, j \in [2 \dots n + 1] : P[i, j] > 0 \implies \\ \left(\neg((i = 2 \wedge j = 2) \vee (i = n + 1 \wedge j = n + 2)) \right) \implies \\ P[i - 1, j] + P[i, j - 1] + P[i + 1, j] + P[i, j + 1] = 2 \end{aligned}$$

2.3.3 Length and turns

Thanks to the modeling decision taken so far, the computation of length and number of turns in the path are straight forward.

- Length $l \in \mathbb{N}$:

$$l = \sum_{i,j=2}^{n+1} P[i, j].$$

- Turn detection is performed using a convolution operation over the non-padded section of the matrix P , with a 2×2 simple filter full of ones.

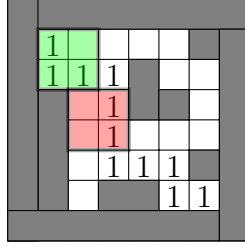


Figure 3: Turn detection in green

The result of each filter application, is encoded in the $n - 1 \times n - 1$ binary matrix tc (turn counter). $\forall i, j \in [2 \dots n]$:

$$tc[i - 1, j - 1] = \begin{cases} 1, & \text{if } \sum_{s,t=0}^1 P[i + s, j + t] \geq 3; \\ 0, & \text{otherwise.} \end{cases}$$

- Turns $t \in \mathbb{N}$:

$$t = \sum_{i,j=1}^{n-1} tc[i, j].$$

2.4 Cost function minimization

Given all the constraint discussed so far, the function f to minimize is the following:

$$f = l^2 + t. \tag{1}$$

The length squared is needed to give priority to the shortest paths and then using turns as tie breaking.

Proof. The proof is based on the fact that:

$$\forall \text{ path } p \text{ of length } l \text{ and turns } t : 1 \leq t < l. \quad (2)$$

Let p_1 be a path of length l_1 and t_1 turns. Let p_2 be a path of length $l_2 = l_1 + 1$ and $t_2 < t_1$ turns.

$$\begin{aligned} f_1 &= l_1^2 + t_1 && [(1)] \\ &< l_1^2 + l_1 && [(2)] \\ &< l_1^2 + 2l_1 && [(2)] \\ &< l_1^2 + 2l_1 + 1 + t_2 && [(2)] \\ &= (l_1 + 1)^2 + t_2 \\ &= l_2^2 + t_2 && [l_2 = l_1 + 1] \\ &= f_2 && [(1)]. \end{aligned}$$

p_1 is preferred because of its length, no matter how many turns p_2 has. \square

2.5 Additional constraints to help the search

Constraints seen so far, together with the minimization of the cost function f , are enough to find the optimal solution. However, during the search procedure, it is possible that undesired candidate solutions are found, for example:



Figure 4: 2×2 and 3×3 loop

Solution containing this kinds of loop will be automatically discarded since they are incrementing the cost function value but, to speed up the search procedure, constraints can be added.

- Avoid 2×2 loop:

$$\forall i, j \in [2 \dots n]: \sum_{s,t=0}^1 P[i+s, j+t] \leq 3.$$

- Avoid 3×3 loop (a zero cell that is not a wall has at most two neighbors):

$$\forall i, j \in [2 \dots n+1] : (P[i, j] = 0 \wedge R[i-1, j-1] = 0) \implies P[i-1, j] + P[i, j-1] + P[i+1, j] + P[i, j+1] \leq 2.$$

- Avoid 4×4 loop (a 2×2 square of zeros without walls has at most 4 neighbors):

$$\begin{aligned} & \forall i, j \in [2 \dots n+1] : \\ & \sum_{s,t=0}^1 P[i+s, j+t] + R[i+s-1, j+t-1] = 0 \implies \\ & P[i-1, j] + P[i-1, j+1] + P[i, j+2] + P[i+1, j+2] + \\ & P[i+2, j] + P[i+2, j+1] + P[i, j-1] + P[i+1, j-1] \leq 4. \end{aligned}$$

Note that there are also other polygons that can be generated and, as a future work, more constraints should be added in order to handle them:

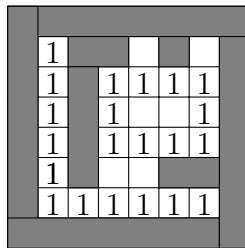


Figure 5: 3×4 rectangle

3 MiniZinc

The formalization of the model done so far can be 1:1 translated into an effective MiniZinc program. In addition, MiniZinc allows to specify which search strategy the solver should follow, this is because for combinatorial problems some search strategies could be more effective than others. For this particular problem, finding the better strategy is not trivial: the goodness of a search strategy really depends on the shape of the room that is randomly generated.

3.1 Search strategy

The strategy listed below achieves good performance with a fairly large and varied sample of rooms.

- Variables: as the key of our problem, it is reasonable to start the search selecting variables from P , since the values of l, tc and t are function of it and they can be easily derived.
- Variable choice: `dom_w_deg` annotation selects the variable of P with the smallest value of domain size divided by weighted degree (which is the number of times it has been in a constraint that caused failure earlier in the search), thus allowing information gathering about the room shape from all the different explorations.
- Constraint choice: `indomain_min` annotation assigns to the selected variable its smallest domain value.

Since $l \ll n^2$ then the number of 0 in P is greater than the number of 1; selecting the lowest value is in general the right choice (reduce backtracking).

- Restart: a bad decisions made at the top of the search tree can take an exponential amount of search to undo. Restart annotation allows to restart the search from the top thus having a chance to make different decisions every time a node limit in the search tree is reached.

With `restart_geometric(n, 1)` the k -th restart has a node limit of $1 \times n^k$. Restart and `dom_w_deg` together are found to be very effective in finding unsatisfiability.

4 Results

$\forall n \in \{6, 8, 10, 12, 14, 16\}$:

- 20 different room instances with $2n$ walls are randomly generated;
- 20 different room instances with n walls are randomly generated.

Results are plotted using a linear scale for the room size (x axis) and a logarithmic scale for the time (y axis). Fixing the room size and the number of walls, the time value that is plotted corresponds to the maximum time that is taken between all the 20 different instances (time in the worst case).

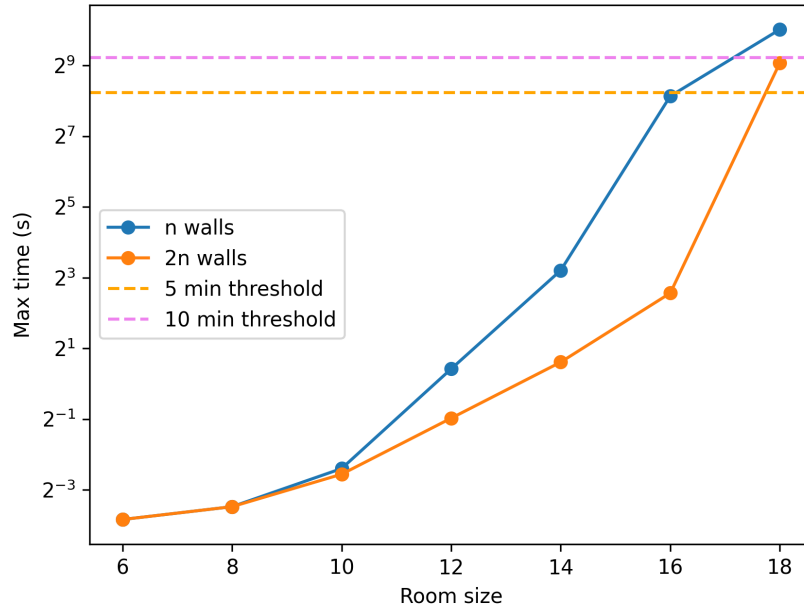


Figure 6: n and $2n$ walls

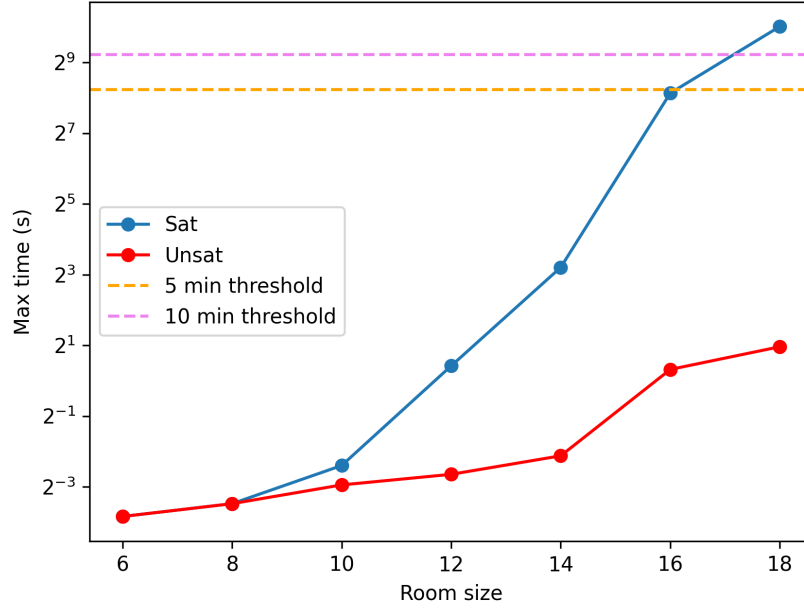


Figure 7: Sat and unsat

From the plots it is clear that finding the solution in a room with $2n$ walls is easier, this is because the more walls there are, the fewer decision variables are involved (thanks to the “avoid walls” static constraint). In general, there are n^2 decision variables with a binary domain; let $w = |\text{walls}|$, then, the search space has a size of 2^{n^2-w} .

An appreciable feature of the model is the ability to quickly find unsatisfiability: this allows the final user to know in a short time whether an admissible solution exists or not and then, if interested, let the solver run in order to find the optimal solution.