Automated Reasoning
Practical Assignment – Part 2

Jelmer Firet (s1023433, jelmer.firet@ru.nl)
Bram Pulles (s1015194, bram.pulles@ru.nl)

December 15, 2022

1. Groups

a) We proved that $I * x = x$, $inv(inv(x)) = x$ and $inv(x) * x = I$ by putting all of the given formulas *literally* in prover9, see `groups_a1.in`.

Using mace4, we show that, in general, $inv(x * y) = inv(x) * inv(y)$ does not hold. The smallest group for which this does not hold has size 6, see `groups_a2.in` and appendix A.

b) Using mace4, we show that the smallest non-Abelian group has size 6, again by *literally* putting in the formulas provided, see `groups_b.in` and appendix A.

c) We use the fact that $x^2 = x * x$, $x^3 = x * (x * x)$ and $x^4 = x * (x * (x * x))$ this makes encoding $x^n = I$ straightforward for $n = 2, 3, 4$.

Using prover9, we show that for $n = 2$ all such groups are Abelian, see `groups_c1.in`.

Using mace4, we show that for $n = 3$ the group is not Abelian in general and the smallest counterexample has size 27, see `groups_c2.in`.

Using mace4, we show that for $n = 4$ the group is not Abelian and the smallest counterexample has size 8, see `groups_c3.in` and appendix A.

2. Robot

2.1. Explanation

In this problem we are given a grid with colored cells. We should program a robot for this grid so that the robot will always find it's way to one of the goal cells if it starts on any start cell. The robot can detect the color of the cell it is on and decide based on that color which direction to move. Our program should find the best assignment of directions to colors. We will Z3 to find this assignment.
In our program we read the grid from a csv file. We then determine the number of colors in this grid. For each color we make a Z3 integer that will store the direction to move for that color. The values of this integer correspond to the following directions:

value	direction	effect on coordinates
0	north	decreasing row
1	east	increasing column
2	south	increasing row
3	west	decreasing column

To ensure that Z3 only assigns one of these, we add the following constraint:

$$\forall_{\texttt{color}} : 0 \leq \texttt{direction}[\texttt{color}] < 4$$

Now we want to evaluate the performance of an assignment. For this we want to know for every cell if we reach a destination, and if so in how many steps. We will add a Z3 integer storing this distance for every cell. For now we will say that this integer has the value ∞ if it can not reach a destination cell.

We then specify how the distances interact with the commands. We will start with the simple cases; the destination and the lava. For a destination cell we do not need to move to reach a destination cell, therefore we add the following constraint:

$$\forall_{0 \leq \texttt{r} < \texttt{R}} \forall_{0 \leq \texttt{c} < \texttt{C}} \text{ if } \texttt{grid}[\texttt{r}][\texttt{c}] = \texttt{DESTINATION} : \texttt{dist}[\texttt{r}][\texttt{c}] = 0$$

We do something similar for lava. From a lava cell we cannot reach a destination cell, so we add the following constraint:

$$\forall_{0 \leq \texttt{r} < \texttt{R}} \forall_{0 \leq \texttt{c} < \texttt{C}} \text{ if } \texttt{grid}[\texttt{r}][\texttt{c}] = \texttt{LAVA} : \texttt{dist}[\texttt{r}][\texttt{c}] = \infty$$

Now we will specify what the robot does for regular cells. For convenience we have a function $\texttt{next}(\texttt{r}, \texttt{c}, \texttt{direction}, \texttt{steps})$ that give the position we go to when we take `steps` steps to `direction` from cell $(\texttt{r}, \texttt{c})$. This function takes into account that the robot stops at the edges of the grid.

$$\begin{aligned} &\forall_{0 \leq \texttt{r} < \texttt{R}} \forall_{0 \leq \texttt{c} < \texttt{C}} \forall_{0 \leq \texttt{dir} < 4} \text{ if } \neg \texttt{special}(\texttt{grid}[\texttt{r}][\texttt{c}]) : \\ &\texttt{direction}[\texttt{grid}[\texttt{r}][\texttt{c}]] = \texttt{dir} \implies \texttt{dist}[\texttt{r}][\texttt{c}] = 1 + \texttt{dist}[\texttt{nr}][\texttt{nc}] \\ &\text{where } (\texttt{nr}, \texttt{nc}) = \texttt{next}(\texttt{r}, \texttt{c}, \texttt{dir}, 1) \end{aligned}$$

Note that we consider $1 + \infty = \infty$. When we give ∞ an actual value, we will have to specify this special case explicitly.
We can now specify how the ice cells work. We want $\texttt{dist}[\texttt{r}][\texttt{c}]$ to store the worst-case distance to a destination cell. We will have a similar condition to the one for the regular cells, but take the maximum of the cells where we can end up. This translates into the following constraint:

$$\forall_{0\leq \texttt{r}<\texttt{R}}\forall_{0\leq \texttt{c}<\texttt{C}}\forall_{0\leq \texttt{dir}<4} \text{ if } \texttt{grid}[\texttt{r}][\texttt{c}] = \texttt{ICE} :$$
$$\texttt{direction}[\texttt{grid}[\texttt{r}][\texttt{c}]] = \texttt{dir} \implies \texttt{dist}[\texttt{r}][\texttt{c}] = 1 + \max(\texttt{dist}[\texttt{nr}][\texttt{nc}], \texttt{dist}[\texttt{nr}', \texttt{nc}'])$$
$$\text{where } (\texttt{nr}, \texttt{nc}) = \texttt{next}(\texttt{r}, \texttt{c}, \texttt{dir}, 1) \text{ and } (\texttt{nr}', \texttt{nc}') = \texttt{next}(\texttt{r}, \texttt{c}, \texttt{dir}, 2)$$

Now we are ready to specify what we want to achieve, we want an assignment of directions to colors that minimizes

$$\max_{0\leq \texttt{r}<\texttt{R}, 0\leq \texttt{c}<\texttt{C}, \texttt{grid}[\texttt{r}][\texttt{c}]=\texttt{START}} \texttt{dist}[\texttt{r}][\texttt{c}]$$

We do this by adding a Z3 integer for this goal and ask the Z3 optimizer to minimize this variable. Then we add the following constraint:

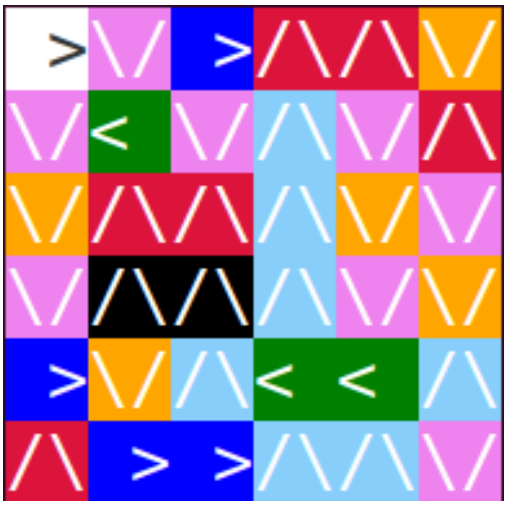
$$\forall_{0\leq \texttt{r}<\texttt{R}}\forall_{0\leq \texttt{c}<\texttt{C}} \text{ if } \texttt{grid}[\texttt{r}][\texttt{c}] = \texttt{START} : \texttt{dist}[\texttt{r}][\texttt{c}] \leq \texttt{goal}$$

Now we can choose a value for ∞. We do not want the $\texttt{dist}[\texttt{r}][\texttt{c}]$ of start cells to be infinite, but otherwise the value does not matter. Therefore we choose the value of ∞ to be $\texttt{goal} + 1$. Then in all of the previous constraints of the form $\texttt{dist}[\texttt{r}][\texttt{c}] = \ldots$ we add the extra option that $\texttt{dist}[\texttt{r}][\texttt{c}] = \texttt{goal} + 1$. This allows Z3 to assign values to all distances even in the case of an infinite loop.

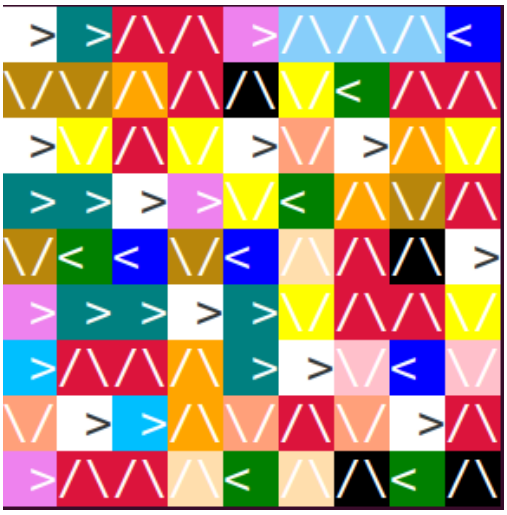
2.2. Results

When we run our program on the demogrid, we find that the minimal solution uses 11 steps[1]. The assignment of directions is visualized in figure 1a.
On grid (a) our program finds that the minimal solution uses 14 steps. The assignment of directions is visualized in figure 1b. On grid (b) our program also finds a minimal solution of 14 steps, which is visualised in figure 1c. On grid (c) our program also finds a minimal solution of 18 steps, which is visualised in figure 1d.

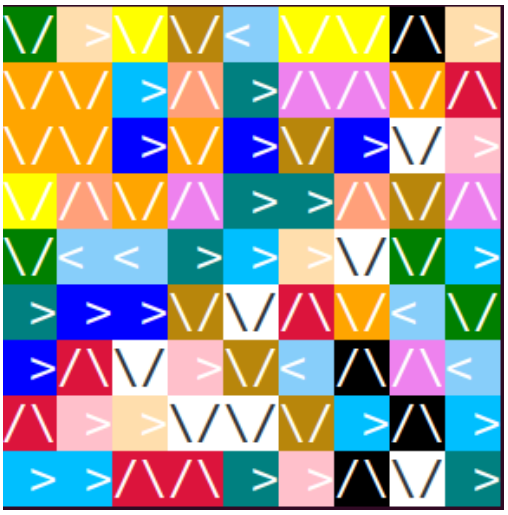
[1]The problem statement gives a "solution for 10 steps", but the image shows a solution with 11 steps



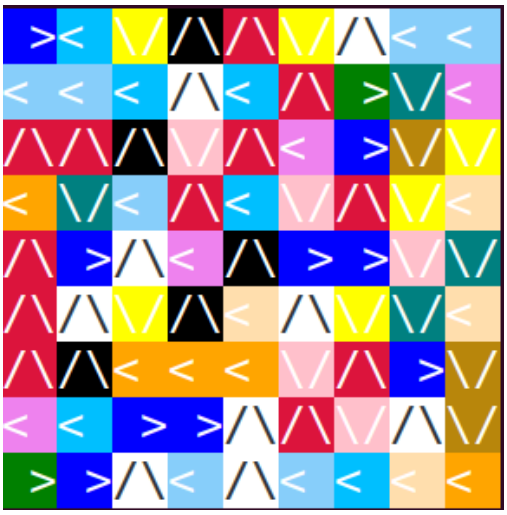
(a) Solution for demo grid



(b) Solution for grid (a)



(c) Solution for grid (b)



(d) Solution for grid (c)

3. Multiset Path Ordering

3.1. Explanation

In this problem we are given a set of inequalities. These inequalities contain terms which are either variables or functions over other terms. Our task is to find a Multiset Path Ordering (MPO), which is an ordering on terms generated from a total quasi-ordering relation $\trianglerighteq$ on the function symbols, using multiset comparison.
To make our lives a bit easier we created a parser which can parse the inequalities. To this end we created four data structures which can represent: terms, functions, variables and inequalities. Notably, every one of these data structures has a function `subterms` which recursively goes down the term and gives all of its decendents. This is useful for when we define the relations on terms.
We define the relations $\trianglerighteq, \triangleright, \equiv$ on functions by creating a new boolean variable for every relation and every pair of functions, i.e. $\trianglerighteq_{\mathtt{fg}}, \triangleright_{\mathtt{fg}}, \equiv_{\mathtt{fg}}$. To get the correct behavior we add the constraints which define these relations, see below. Note, that the relation $\leq$ is purely to prevent duplicates, such as $\mathtt{f} \trianglerighteq \mathtt{g} \vee \mathtt{g} \trianglerighteq \mathtt{f}$ *and* $\mathtt{g} \trianglerighteq \mathtt{f} \vee \mathtt{f} \trianglerighteq \mathtt{g}$.

$$\forall_{\mathtt{f}} \forall_{\mathtt{f} \leq \mathtt{g}} \trianglerighteq_{\mathtt{fg}} \vee \trianglerighteq_{\mathtt{gf}}$$

$$\forall_{\mathtt{f}} \forall_{\mathtt{f} \leq \mathtt{g}} \equiv_{\mathtt{fg}} = \trianglerighteq_{\mathtt{fg}} \wedge \trianglerighteq_{\mathtt{gf}}$$

$$\forall_{\mathtt{f}} \trianglerighteq_{\mathtt{ff}}$$

$$\forall_{\mathtt{f}} \forall_{\mathtt{g}} \triangleright_{\mathtt{fg}} = \trianglerighteq_{\mathtt{fg}} \wedge \neg \trianglerighteq_{\mathtt{gf}}$$

$$\forall_{\mathtt{f}} \forall_{\mathtt{g}} \forall_{\mathtt{h}} \ \trianglerighteq_{\mathtt{fg}} \wedge \trianglerighteq_{\mathtt{gh}} \rightarrow \trianglerighteq_{\mathtt{fh}}$$

We define the toprelations $\gtrsim, \succ, \approx$ on terms by creating a new variable for every relation and every pair of terms, i.e. $\gtrsim_{st}, \succ_{st}, \approx_{st}$. To keep everything comprehensible we subdivide the relations $\succ, \approx$ and create new variables $\succ_{a,st}, \succ_{b,st}, \succ_{c,st}, \approx_{a,st}, \approx_{b,st}$. The toprelations are now defined using the constraints shown below.

$$\forall_s \forall_t \gtrsim_{st} = \succ_{st} \vee \approx_{st}$$
$$\forall_s \forall_t \succ_{st} = \succ_{a,st} \vee \succ_{b,st} \vee \succ_{c,st}$$
$$\forall_s \forall_t \approx_{st} = \approx_{a,st} \vee \approx_{b,st}$$

We define constraints for when the types do not match. For example, when either s or t is a function we restrict $\neg \approx_{a,st}$. We do the same for all the other subrelations, these constraints follow trivially and will not be written out. The constraints for when the types do match are nontrivial. The simple subrelation constraints are as follows.

$$\forall_{s=\mathtt{f}(s_1,\ldots,s_n)}\forall_t \succ_{a,st} = \bigvee_{s_i} \succsim_{s_i,t}$$

$$\forall_{s=\mathtt{f}(s_1,\ldots,s_n)}\forall_{t=\mathtt{g}(t_1,\ldots,t_m)} \succ_{b,st} = \rhd_{\mathtt{f},\mathtt{g}} \wedge \bigwedge_{t_i} \succ_{s,t_i}$$

$$\forall_s\forall_t \approx_{a,st} = \approx_{st}$$

We are now left with the two constraints which use the multiset properties. We will first take a look at $\approx_{b,st}$. Note, when s or t is not a function, or the number of arguments in s and t are not the same we have $\neg \approx_{b,st}$. For all of the other cases, where $s = \mathtt{f}(s_1, \ldots, t_n)$ and $t = \mathtt{g}(t_1, \ldots, t_n)$, we check whether there exists a permutation of the arguments of t such that $\forall_n \approx_{s_i,t_i}$. We define $\mathcal{P}(t_1, \ldots, t_n)$ to give all the possible permutations and use t_i^p to denote t_i in permutation $p \in \mathcal{P}$.

$$\forall_{s=\mathtt{f}(s_1,\ldots,s_n)}\forall_{t=\mathtt{g}(t_1,\ldots,t_m)} \approx_{b,st} = \equiv_{\mathtt{f},\mathtt{g}} \wedge \bigvee_{p\in\mathcal{P}(t_1,\ldots,t_n)} \bigwedge_i \approx_{s_i,t_i^p}$$

The final constraint $\succ_{c,st}$ is the most elaborate of all. We use the tip provided in the assignment. We generate all the possible $\mathtt{EQ}$ and φ, such that $\exists_{i\in\{1,\ldots,n\}}\ \mathtt{EQ}(i) = \bot$ and $\forall_i\ \mathtt{EQ}(\varphi(i)) = \top \to \forall_{j\neq i}\ \varphi(i) \neq \varphi(j)$. We can do this by generating all $\mathtt{EQ}$ and φ, then filtering out the functions which do not satisfy the given constraints. We define $\mathcal{P}_\varphi^{\mathtt{EQ}}(n, m)$ to give all the $\mathtt{EQ}, \varphi$ pairs which satisfy the given constraints. The final constraint is given below, seperated for when $\mathtt{EQ}(\varphi(i)) = \bot$ or $\mathtt{EQ}(\varphi(i)) = \top$.

$$\forall_{s=\mathtt{f}(s_1,\ldots,s_n)}\forall_{t=\mathtt{g}(t_1,\ldots,t_m)} \succ_{c,st} = \equiv_{\mathtt{f},\mathtt{g}} \wedge \bigvee_{\mathtt{EQ},\varphi\in\mathcal{P}_\varphi^{\mathtt{EQ}}(n,m)} \left(\bigwedge_{i,\mathtt{EQ}(\varphi(i))=\bot} \succ_{s_{\varphi(i)},t_i} \bigwedge_{i,\mathtt{EQ}(\varphi(i))=\top} \approx_{s_{\varphi(i)},t_i} \right)$$

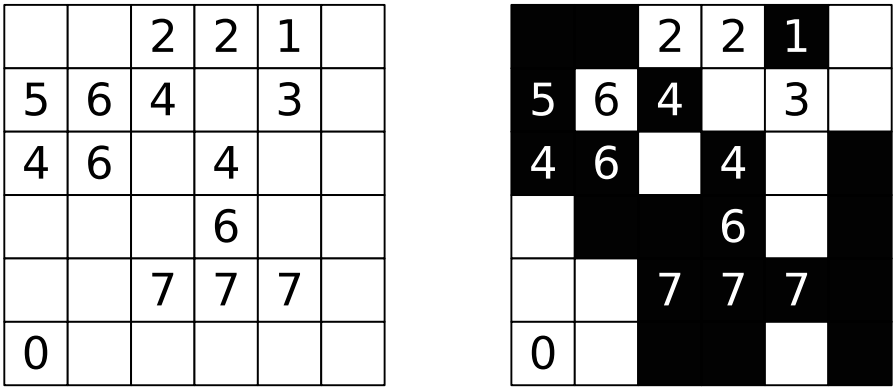
3.2. Results

Running Z3 with all the constraints and variable definitions as given above provides us with the following ordering: $\mathtt{a} \rhd \mathtt{c} \rhd \mathtt{f} \rhd \mathtt{g} \rhd \mathtt{b} \rhd \mathtt{h} \rhd \mathtt{d}$. Finding another symbol ordering is rather easy, we just added the constraint $\neg\rhd_{\mathtt{h},\mathtt{d}}$. This gives us a new symbol ordering: $\mathtt{a} \rhd \mathtt{c} \rhd \mathtt{b} \equiv \mathtt{f} \rhd \mathtt{g} \rhd \mathtt{d} \rhd \mathtt{h}$.
The human-readable proof for the inequality $\mathtt{h}(\mathtt{g}(x, \mathtt{g}(u, z)), \mathtt{c}(x, y, x, z)) \rhd \mathtt{f}(\mathtt{d}(x, z), u)$ given the first ordering is as follows. By 2b it suffices to proof that $\mathtt{h} \rhd \mathtt{d}$ which holds, and $\mathtt{h}(\mathtt{g}(x, \mathtt{g}(u, z)), \mathtt{c}(x, y, x, z)) \succ x$, and $\mathtt{h}(\mathtt{g}(x, \mathtt{g}(u, z)), \mathtt{c}(x, y, x, z)) \succ z$. Now these two inequalities trivially follow from recursively applying rule 2a until a variable x or z is reached inside the arguments of $\mathtt{h}$, using 3a on the variables completes the proof.

4. Mosaic

4.1. Puzzle

Our problem is to generate a minimal mosaic puzzle. Mosaic[2] is a relatively simple puzzle. You are given a grid of squares in which some cells have a number. You should color each cell either white or black. A number indicates how many cells in the surrounding 3×3 should be colored black. Below you can find an example of a mosaic puzzle and its solution.



The puzzle we generate should satisfy the following requirements:

1. The puzzle has an unique solution.
2. This solution should match the solution we provide as input.
3. There are no other puzzles with the same solution that have fewer numbers given.

We have made two programs to generate puzzles. The first uses a SMT solver iteratively to find constraints that are necessary for a puzzle to be unique. Then it runs another SMT solver to minimize the number of hints. The other approach uses the unsat core. While Z3 finds a small puzzle using this method, it is not guaranteed to satisfy (3).

4.2. Generation using unsat core

Since we want to use the unsat core, we need to make a set of constraints that is not satisfiable. We will use this unsatisfiability to guarantee requirement (1), namely we will try to find a second solution.
For this we add a variable for each cell, whether it is black in the second solution or not. Since we will be adding these later, we encode this as a 0-1 constrained Z3 integer:

$$\forall_r, c : 0 \leq \text{filled}_{r,c} \leq 1$$

[2] Also known as: fill-a-pix

Then we will add the constraint that at least one of those variable does not match the given solution (grid), so that we force Z3 to look for a second solution:

$$\exists_{r,c} : \text{filled}_{r,c} \neq \text{grid}_{r,c}$$

We will ask Z3 to find this second solution to the puzzle with a number in every cell. Since we have a desired solution, we can determine which value the numbers should have. For every cell we add the constraint that it has the same number of black cells surrounding it in both solutions:

$$\forall_{r,c} : \sum_{\substack{r-1 \leq r' \leq r+1 \\ c-1 \leq c' \leq c+1}} \text{grid}_{r',c'} = \sum_{\substack{r-1 \leq r' \leq r+1 \\ c-1 \leq c' \leq c+1}} \text{filled}_{r',c'}$$

This formula does not handle the edges, but we do handle those cases in our code. Since we are interested in the positions of the numbers in the puzzle, we track these constraints so that Z3 will give them as part of the unsat core.
Now that we have all constraints we ask Z3 to check if there is a solution. If there is a solution we print this alternate solution. There exist desired solutions for which this happens, but it is rare.
Otherwise we ask Z3 for an unsat core. This gives us a list of the number constraints that guarantee there is no second solution. Since these constraints correspond with numbers in the grid, we have found a puzzle that satisfies requirements (1) and (2). We enabled the `core.minimize` flag of Z3 to try find a minimal puzzle. However the minimal unsat core of Z3 is "best-effort", and there is no guarantee that it is optimal. In particular, the example puzzle above has been generated with this program and has 15 numbers. However, our other program has found the following puzzle with 13 hints:

		2	2	1	
5	6				
4		6	4		
					4
		7			
0			5		3

■	■	2	2	1	
5	6	■			
4	■	6	4		■
	■	■	■		4
	■	7	■	■	■
0		■	5		3

4.3. Generation using iteration

In this approach we use two Z3 optimizers. The first optimizer searches for constraints on where to place numbers and makes these constraints as small as possible. The second optimizer tries to find the smallest placement of numbers that satisfies the constraints

generated by the first optimizer.

The idea of the first optimizer that we try to find a puzzle that satisfies all the generated constraints but still has a different solution. Then we consider which hints can differentiate both solutions. These are exactly the cells where the count of surrounding black squares differ. Any puzzle with the desired solution as it's only solution must then have a number in at least one of those squares.

The constraints we add to the first optimizer largely correspond with what we did for the unsat core. However, we now have three types of variables:

hinted	a puzzle that satisfies the generated constraints
filled	an alternate solution to that puzzle
contributes	places where the count of surrounding black squares differ.

Again we encode these variables with a 0-1 constrained integer in Z3:

$$\forall_{r,c} : 0 \leq \text{hinted}_{r,c} \leq 1$$

$$\forall_{r,c} : 0 \leq \text{filled}_{r,c} \leq 1$$

$$\forall_{r,c} : 0 \leq \text{contributes}_{r,c} \leq 1$$

We replace the number constraint with a new one that takes into account whether a number is placed into a cell:

$$\forall_{r,c} : \text{hinted}_{r,c} \rightarrow \sum_{\substack{r-1 \leq r' \leq r+1 \\ c-1 \leq c' \leq c+1}} \text{grid}_{r',c'} = \sum_{\substack{r-1 \leq r' \leq r+1 \\ c-1 \leq c' \leq c+1}} \text{filled}_{r',c'}$$

As before we require that the alternate solution differs in at least one cell from the desired solution:

$$\exists_{r,c} : \text{filled}_{r,c} \neq \text{grid}_{r,c}$$

Next we encode that a cell differentiates both solutions when the black counts differ:

$$\forall_{r,c} : \sum_{\substack{r-1 \leq r' \leq r+1 \\ c-1 \leq c' \leq c+1}} \text{grid}_{r',c'} = \sum_{\substack{r-1 \leq r' \leq r+1 \\ c-1 \leq c' \leq c+1}} \text{filled}_{r',c'} \rightarrow \text{contributes}_{r,c} = 1$$

The clause we generate has a literal for every cell that distinguishes both solutions. Since SMT solvers are generally faster on smaller clauses, we minimize the number of cells that distinguish both solutions.

$$\sum_{r,c} \text{constributes}_{r,c} = \min(\sum_{r,c} \text{contributes}_{r,c})$$

When we run the first optimizer, we can have two results. If the solver returns unsat, there is no puzzle that satisfies all generated constraints and has multiple solutions. In other words, all puzzles satisfying the generated constraints have a unique solution. We

then use the second optimizer to find the smallest puzzle satisfying all generated constraints.

In the other case the solver provides a model that includes a puzzle, an alternate solution and the places where the black counts differ. We will add a new clause to both optimizers that encodes that a puzzle should distinguish both solutions:

$$\exists_{r,c:\text{model}(\text{contributes}_{r,c})=1} : \text{hinted}_{r,c}$$

After adding this constraint we run the first optimizer again to see if there are still alternate solutions.

The second optimizer is a lot easier. It only uses the hinted variables, and the constraint that these are 0 or 1. It also includes all generated constraints to guarantee that the resulting puzzle has an unique solution. Finally it has the goal to minimize the number of cells that are hinted.

This optimizer will always give a model[3]. This model corresponds to a minimal puzzle with a unique solution. Suppose there is another puzzle that has fewer hints. Then this puzzle does not satisfy at least one of the generated constraints. But then the alternate solution we found in the step that generated this constraint is also an alternate solution to the puzzle. Therefore there are no smaller puzzles with a unique solution.

4.4. Results

We have discussed two approaches to generating minimal mosaic puzzles for a given solution. The unsat-core approach is more efficient since it uses features of Z3 directly. We were able to generate puzzles with a 20×20 grid within a couple of seconds. Unfortunately there is no guarantee that the generated puzzle is minimal, as we have shown with the example above.

The iterative approach is quite slow. It took multiple minutes to find the puzzle that disproved minimality of the unsat-core puzzle. The largest puzzle we have been able to generate is 8×8, and took a couple of hours of computation. We are however confident that it generates globally minimal puzzles. An added benefit of this approach is that it defines the space of unique puzzles, so we can easily adapt the goal to find unique puzzles with some other property. For example, we can look for a minimal puzzle that has no adjacent hints, something we cannot encode in the unsat-core approach.

[3]Except in the rare cases where even the puzzle with a number in every cell has multiple solutions

A. Groups

a) Smallest finite group for which $inv(x * y) = inv(x) * inv(y)$ does not hold.

```
interpretation( 6, [number=1, seconds=0], [

    function(I, [ 0 ]),

    function(c1, [ 1 ]),

    function(c2, [ 2 ]),

    function(inv(_), [ 0, 1, 2, 4, 3, 5 ]),

    function(*(_,_), [
        0, 1, 2, 3, 4, 5,
        1, 0, 3, 2, 5, 4,
        2, 4, 0, 5, 1, 3,
        3, 5, 1, 4, 0, 2,
        4, 2, 5, 0, 3, 1,
        5, 3, 4, 1, 2, 0 ])
]).
```

b) Smallest non-abelian group.

```
interpretation( 6, [number=1, seconds=0], [

    function(I, [ 0 ]),

    function(c1, [ 1 ]),

    function(c2, [ 2 ]),

    function(inv(_), [ 0, 1, 2, 4, 3, 5 ]),

    function(*(_,_), [
        0, 1, 2, 3, 4, 5,
        1, 0, 3, 2, 5, 4,
        2, 4, 0, 5, 1, 3,
        3, 5, 1, 4, 0, 2,
        4, 2, 5, 0, 3, 1,
        5, 3, 4, 1, 2, 0 ])
]).
```

c) Smallest non-abelian group with $x^4 = I$.

```
interpretation( 6, [number=1, seconds=0], [

    function(I, [ 0 ]),

    function(c1, [ 1 ]),

    function(c2, [ 2 ]),

    function(inv(_), [ 0, 1, 2, 4, 3, 5, 6, 7 ]),

    function(*(_,_), [
        0, 1, 2, 3, 4, 5, 6, 7,
        1, 0, 3, 2, 5, 4, 7, 6,
        2, 4, 0, 6, 1, 7, 3, 5,
        3, 5, 1, 7, 0, 6, 2, 4,
        4, 2, 6, 0, 7, 1, 5, 3,
        5, 3, 7, 1, 6, 0, 4, 2,
        6, 7, 4, 5, 2, 3, 0, 1,
        7, 6, 5, 4, 3, 2, 1, 0 ])
]).
```