# Assignment 3: Solve Nonograms

AI-1 Systems Project (Winter Semester 2023/2024)

Jan Frederik Schaefer

Friedrich-Alexander-Universität Erlangen-Nürnberg, Department Informatik

| | |
|---|---|
| Topic: | SAT Solvers |
| Due on: | March 15, 2024 |
| Version from: | January 11, 2024 |
| Author: | Jan Frederik Schaefer |

**Summary** Solve picture puzzles using a SAT solver. The puzzles are a generalization of **nonograms** [WN], which are NP-complete logic puzzles originating in Japan. Besides the traditional rectangular grids, we will also cover hexagonal grids. You will have to convert the puzzle to a SAT problem, solve it with a SAT solver, and then use the solution to the SAT problem to infer the correct coloring of the cells.

This assignment is more work than the other assignments and is therefore worth more points than the usual 100 and has a later deadline. Furthermore, you are supposed to also compare different approaches (see Section 3 and Appendix B).

### Objectives

1. Gain some experience with encoding a problem as a SAT problem,
2. understand that/why some encodings scale better than others.

### Prerequisites and useful methods

1. SAT solving (as discussed in the AI lecture, though you should use an existing solver, i.e. the algorithms for SAT solving are not relevant),
2. Boolean/propositional logic (De Morgan's law, material implication, DNF, CNF, . . . ).

Copyright © 2021–2024 Jan Frederik Schaefer, Michael Kohlhase.

# Contents


**1 Puzzle rules** **3**

**2 Nonogram formats and SAT solvers** **5**

2.1 Puzzle Format . . . . . . . . . . 5

2.2 Storing, visualizing and checking the results . . . . . . . . . . 5

2.3 Using SAT solvers . . . . . . . . . . 6

**3 Comparing different approaches** **7**

**4 Submission** **7**

**5 A few tips** **8**

**6 Points** **9**

**A Translating to CNF: Tips and exercises (optional)** **10**

A.1 Exercises . . . . . . . . . . 10

A.2 Using helper variables . . . . . . . . . . 10

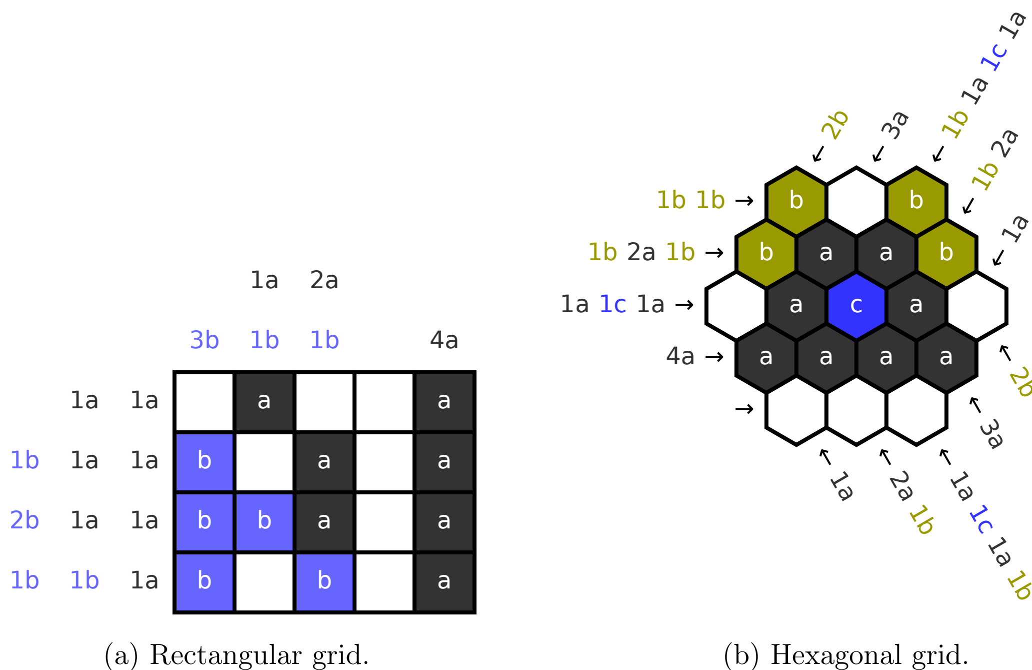
**B Encoding nonograms as SAT problems** **11**

B.1 Approach 1: Listing possible arrangements . . . . . . . . . . 12

B.2 Approach 2: Use variables to denote each block start . . . . . . . . . . 13

B.3 Approach 3: Enclose blocks . . . . . . . . . . 14

B.4 Approach 4: Make an automaton . . . . . . . . . . 15




(a) Rectangular grid.

(b) Hexagonal grid.

Figure 1: Two example generalized nonograms (solved).

# 1 Puzzle rules

The goal of a (generalized) nonogram is to color the cells of a grid using clues. A **clue** is a sequence $n_1c_1 \dots n_kc_k$, where $n_i$ are numbers and $c_i$ are color references. Each element $n_ic_i$ indicates a continuous **block** of $n_i$ cells of color $c_i$. The blocks occur in the same order as listed in the clue. Consecutive blocks of the same color must be separated by at least one empty cell.

As an example, let us take a look at the third row of the example nonogram in Figure 1a. We use letters (here `a` and `b`) to refer to colors. The clue `2b 1a 1a` for the third row indicates that it starts with a 2-cell block of color `b` (in this case blue), which is followed by two 1-cell blocks of color `a` (black). Since the last two blocks have the same color, they must be separated by at least one empty cell.

Sometimes, clues have incomplete information: It is possible that the length of a block is not specified. Instead, a "`+`" is used to indicate that the block is at least of length 1.

**Simplification:** To make the assignment a bit easier, we will only use nonograms with 1 or 2 colors.

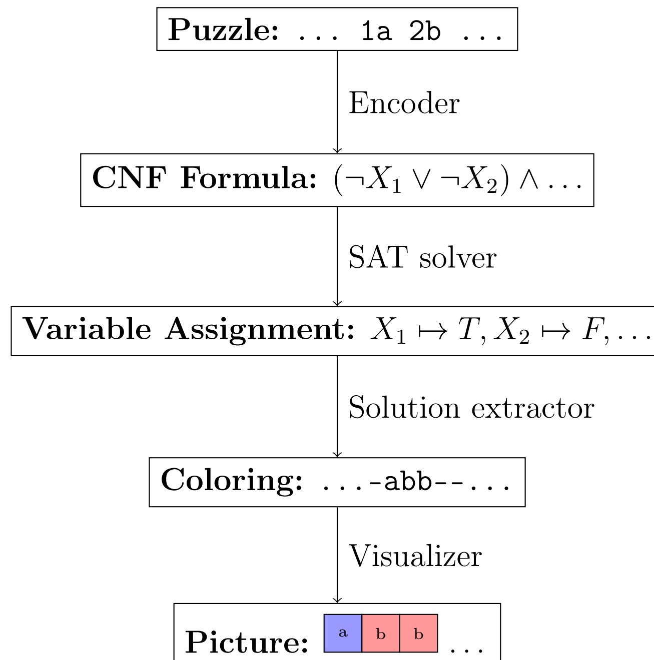



Figure 2: A sequence of steps to solve a nonogram using a SAT solver.

```
rect 4 5
#ffffff #333333 #6666ff
1a 1a
1b 1a 1a
2b 1a 1a
1b 1b 1a
3b
1a 1b
2a 1b

4a
```

```
hex 3
#ffffff #333333 #999900 #3333ff
1b 1b
1b 2a 1b
1a 1c 1a
4a

1a
2a 1b
1a 1c 1a 1b
3a
2b
1a
1b 2a
1b 1a 1c 1a
3a
2b
```

Listing 1: Encoding of the nonograms from Figure 1a (left) and Figure 1b (right).

# 2 Nonogram formats and SAT solvers

This section describes how generalized nonograms and their solutions should be represented. We also discuss the basics of using a SAT solver. Figure 2 sketches the pipeline of a solver: You start with an unsolved nonogram as an input (the format is specified in Section 2.1). The main challenge for you will be to translate it into a CNF formula that can be solved with a SAT solver. Appendix B sketches different ways how this can be done. A SAT solver (Section 2.3) can provide a variable assignment that satisfies the formula. If you chose a good encoding, it should be easy to infer a solution for the nonogram (i.e. a coloring) from the variable assignment. Section 2.2 specifies a format for storing the solution so that it can be checked automatically. In the following sections, we will explore these steps in more detail.

## 2.1 Puzzle Format

Puzzles are represented as text files. First, the grid is described:

- Rectangular grids start with the line `rect <height> <width>`, where `<height>` is the number of rows and `<width>` is the number of columns.
- Hexagonal grids start with the line `hex <size>`, where `<size>` indicates the side length of the hexagon.

The second line lists the colors, starting with the background color, followed by the colors that are afterwards referred to as `a`, `b`, `c`, etc.

Each of the remaining lines corresponds to a clue. For a rectangular grid, the row clues are listed first, followed by the column clues. In a hexagonal grid, we have clues in three directions (see Figure 1b). The file lists them counter-clockwise, starting with the top-left corner (`1b 1b` in the example). Listing 1 shows the encodings of the example nonograms.

## 2.2 Storing, visualizing and checking the results

The solution of a nonogram should be represented as a text file. Each line describes the coloring of one row of cells. Uncolored cells are represented with a `-` and colored cells with the name of the color (`a`, `b`, ...). Listing 2 shows the solutions to the example nonograms from Figure 1.

The assignment repository [AR] contains a script that you can use to check if the solutions are correct. It also contains a script for visualizing the solution similar to the visualizations

```
-a--a
b-a-a
bba-a
b b-a
```

```
b-b
baab
-aca-
aaaa
---
```

Listing 2: Solution files for the nonograms from Figure 1a (left) and Figure 1b (right).

```
p cnf 3 2
1 2 0
-1 3 0
```

```
-1 2 -3
```

Listing 3: Encoding of $(X_1 \vee X_2) \wedge (\neg X_1 \vee X_3)$ (left) and the solution $X_1 = F, X_2 = T, X_3 = F$ (right).

in Figure 1. More details are provided in the README of the assignment repository.

## 2.3 Using SAT solvers

To solve the nonograms, you should translate them into a SAT problem that can be solved by off-the-shelf SAT solvers.

We will represent SAT problems using (a subset of) the DIMACS format, which is commonly used for SAT competitions. Listing 3 shows the encoding of an example formula. The first line is always of the form `p cnf <nvars> <nclauses>`, where `<nvars>` is the number of variables used and `<nclauses>` is the number of clauses. Each subsequent line encodes one clause as a list of integers. Positive integers denote variables and negative integers their negations (i.e. the literal $X_i$ is denoted by $i$ and the literal $\neg X_i$ by $-i$). The end of a clause is marked by a 0. Problems in that format can be solved by many SAT solvers, including MiniSat [MS], which is relatively easy to use and install, and Kissat [KS], which has won recent SAT competitions (see e.g. [S22]).

Variable assignments that satisfy the problem are also encoded as a list of integers, where positive integers indicate variables that are true and negative integers indicate variables set to false (see Listing 3 for an example). Both MiniSat and Kissat produce solutions in that format.

# 3 Comparing different approaches

This assignment is a bit different from the other ones. Encoding the nonograms as a SAT problem is not easy, especially if you are not used to working with a SAT solver. In the past, students typically came up with a very inefficient encoding, which did not scale to larger problems, and various optimizations could not improve the efficiency on a fundamental level. To avoid this, we sketch different approaches how the nonograms can be encoded as a SAT problem in Appendix B. We would like you to estimate how well these approaches would scale to larger nonograms. You do not have to indicate a very precise complexity class for each of them (that is a bit tricky in some cases, though it is great if you manage), but you should determine which ones would scale better (and why). You should also implement two of the approaches (for rectangular single-color nonograms) and measure how efficient they are. The assignment repository [AR] contains nonograms of different sizes that you can use for the measurements. If you come up with your own approach, you can of course implement that one instead. You are also welcome to tweak the suggested approaches in any way you see fit.

Your complexity estimates and measurements should be part of your evaluation. As this makes the evaluation-part of the submission longer and more work, it will also be worth more points (see Section 6).

# 4 Submission

At the deadline, we will download a snapshot of your repository. It should contain:

1. All your code.
2. Solutions to all the example nonograms in the assignment repository [AR]. That means that you should have a solution file `<name>.solution` as described in Section 2.2 for every problem `<name>.clues`.
3. A README file explaining how to run your code to solve a new nonogram.
4. An evaluation of your solution as a PDF that describes what worked well/didn't work well when you tried to solve the problem. In this case, it should also include estimates of how efficient the different approaches explained in Appendix B are and a plot with measurements for the efficiency of two different approaches (see Section 3). A typical evaluation should be 1–3 pages long, but it is okay if your evaluation is much longer (it would just be more work from you than we expect).

# 5 A few tips

1. For debugging purposes, you might want to make your own, small nonograms (e.g. $2 \times 2$ or $2 \times 3$ rectangles). If the small nonograms work, but you still have problems with the larger ones, debugging gets trickier. Here are two strategies for re-producing bugs on a smaller problem if the nonogram is rectangular:
   (a) *Case 1:* You do not have enough constraints, i.e. you find a solution that should be impossible according to the clues. In this case, you can reduce the problem to the problematic row/column. Let us assume that the issue is in a row. Then you can make a new nonogram that only consists of that row, using the same row clue, but enforcing the wrong solution with custom column hints. This should be unsatisfiable, but (because of the bug), you should get a solution anyway. You can then carefully examine the solution and find out what constraint was missing/did not work.
   (b) *Case 2:* You have too many constraints, i.e. you do not find any solution, even though the problem should be solvable. This strategy requires you to know the correct solution. You can test each row/column separately. To test a row, you provide the row clue and use the column clues to enforce the correct solution. This should be satisfiable, but for at least one row, or column, it should be unsatisfiable because of the bug. Once you have identified the problematic row/column, you can try to examine what constraint prevents the correct solution.
2. As a starting point, you could focus on classical nonograms (rectangular grid and just one non-background color). When you have a good understanding of how to solve them, moving on to the other nonograms should be more manageable.
3. People usually expect that the SAT solver will be the bottleneck. But for very inefficient encodings, it is more common that the encoder becomes the bottleneck. Additionally, it appears that SAT solvers tend to process more compact encodings faster.
4. This assignment benefits from discussion. You should ask for help, e.g. at the office hours or in the assignment room on Matrix – especially if you are stuck at the early stages (which happens to many students).
5. Have a lot of fun :-)

## 6 Points

This assignment is worth 120 points. 90 points are awarded for the solutions to the nonograms in the assignment repository. Concretely, you will get $\lceil 18\sqrt{n} \rceil$ points where $n$ is the number of correctly solved nonograms (there are 25 in total). The remaining 30 points are awarded for the quality of your submission, the README, and the evaluation and comparison of approaches (see Section 3).

If the grading scheme does not seem to work well, we might adjust it later on (likely in your favor).

## References


[AR] *Repository for Assignment 3: Solve Nonograms*. URL: `https://gitlab.rrze.fau.de/wrv/AISysProj/ws2324/a1.3-solve-nonograms/assignment`.

[KS] *Kissat SAT Solver*. URL: `http://fmv.jku.at/kissat/` (visited on 01/18/2022).

[MS] *The MiniSat Page*. URL: `http://minisat.se/` (visited on 01/18/2022).

[S22] *SAT Competition 2022: Results*. URL: `https://satcompetition.github.io/2022/results.html` (visited on 07/21/2023).

[WN] *Nonogram*. URL: `https://en.wikipedia.org/wiki/Nonogram` (visited on 01/18/2022).

# A Translating to CNF: Tips and exercises (optional)

SAT solvers expect formulae in CNF. When you try to encode the nonogram as a SAT problem, you might come up with formulae that are not in CNF. To help you get started, we thought it might be useful if we list some formulae for you to convert to CNF as an exercise. Your conversions should be efficient (otherwise, your nonogram encoding might not scale).

## A.1 Exercises

Here are some formulae for you to convert to CNF in increasing difficulty:

1. $A \Rightarrow B$
2. $A \Rightarrow (B \vee C)$
3. $A \Leftrightarrow B$
4. $(\neg A) \Rightarrow (B \vee \neg C \vee D)$
5. $A \Rightarrow (B \wedge C)$
6. $A \Rightarrow (B \wedge C \wedge D \wedge E)$
7. $(A \wedge B) \Rightarrow C$
8. $(A \wedge B \wedge C \wedge D) \Rightarrow E$
9. $(A \wedge B \wedge C \wedge D) \Rightarrow (E \vee F \vee G)$

For the following formulae, it helps to introduce helper variables (see also next section):

1. $(A \vee B \vee C \vee D) \Rightarrow (E \wedge F \wedge G \wedge H)$
2. $(A \wedge B \wedge C) \vee (D \wedge E \wedge F) \vee (G \wedge H \wedge I)$

## A.2 Using helper variables

Sometimes, it is helpful to introduce further **helper variables** to make the translation more efficient. For example, the formula $\varphi := (X \wedge Y) \vee Z$ could be translated to the CNF formula $\psi := (A \vee Z) \wedge (\neg A \vee X) \wedge (\neg A \vee Y)$. While $\varphi$ and $\psi$ are technically not equivalent, they are equisatisfiable. More concretely: for any assignment that satisfies $\varphi$, we can find an assignment that also satisfies $\psi$. Furthermore, any assignment that satisfies $\psi$ also satisfies $\varphi$.

To show you a more relevant example, let us consider the formula

$$(X_1 \vee X_2 \vee X_3 \vee X_4) \Rightarrow (Y_1 \wedge Y_2 \wedge Y_3 \wedge Y_4)$$

We can see that whenever $X_1$ is true, $Y_1$ must also be true ($X_1 \Rightarrow Y_1$). Similarly, $X_1 \Rightarrow Y_2$, $X_1 \Rightarrow Y_3$ and $X_1 \Rightarrow Y_4$. Furthermore, $X_2 \Rightarrow Y_1$ and so on. Since $A \Rightarrow B$ is the same as

$\neg A \vee B$, we can translate the formula to the following CNF formula[1]:

$$
\begin{aligned}
&(\neg X_1 \vee Y_1) \wedge (\neg X_1 \vee Y_2) \wedge (\neg X_1 \vee Y_3) \wedge (\neg X_1 \vee Y_4) \wedge \\
&(\neg X_2 \vee Y_1) \wedge (\neg X_2 \vee Y_2) \wedge (\neg X_2 \vee Y_3) \wedge (\neg X_2 \vee Y_4) \wedge \\
&(\neg X_3 \vee Y_1) \wedge (\neg X_3 \vee Y_2) \wedge (\neg X_3 \vee Y_3) \wedge (\neg X_3 \vee Y_4) \wedge \\
&(\neg X_4 \vee Y_1) \wedge (\neg X_4 \vee Y_2) \wedge (\neg X_4 \vee Y_3) \wedge (\neg X_4 \vee Y_4)
\end{aligned}
$$

This is not particularly efficient: If we generalize the original formula to $(X_1 \vee \ldots \vee X_n) \Rightarrow (Y_1 \vee \ldots \vee Y_n)$, we would get $n^2$ clauses. We can reduce this to $2n + 1$ if we introduce two helper variables $L$ and $R$, where $L$ is intuitively the antecedent of the implication (the left hand side) and $R$ is the consequent (the right hand side). That lets us re-write the formula as $L \Rightarrow R$. Now, if $X_1$ is true, then the antecedent must be true, i.e. $X_1 \Rightarrow L$. Similarly, $X_2 \Rightarrow L$ and so on. And if the consequent, i.e. $R$, is true, then $Y_1$ must be true ($R \Rightarrow Y_1$). Similarly, $R \Rightarrow Y_2$ and so on. So we get the equisatisfiable formula[2]

$$
\begin{aligned}
&(\neg L \vee R) \wedge \\
&(\neg X_1 \vee L) \wedge (\neg X_2 \vee L) \wedge (\neg X_3 \vee L) \wedge (\neg X_4 \vee L) \wedge \\
&(\neg R \vee Y_1) \wedge (\neg R \vee Y_2) \wedge (\neg R \vee Y_3) \wedge (\neg R \vee Y_4)
\end{aligned}
$$

This approach is similar to the Tseytin transformation[1]. EdN:1

# B Encoding nonograms as SAT problems

In this appendix, we will sketch different ways how a traditional nonogram can be encoded has a SAT problem. Some are easier to implement than others and some perform much better than others (at least for large nonograms). For didactic reasons, we will leave the analysis for you (see Section 3) and only sketch the approach.

As a running example, we will explore the encoding of a single clue, `2a 1a 2a` for a 10-cell row in a traditional (i.e. single-colored) nonogram. We will use the variables $C_1, \ldots, C_{10}$ to indicate for each cell if it should be colored. When we get a variable assignment from the SAT solver, we can simply check what values are assigned to $C_1, \ldots, C_{10}$ to know how we

[1] Actually, I only argued that the following formula must be true if the original formula is true. For equivalence, we need the other direction as well. If you stare at it for a while, I hope that you would agree that the other direction works as well.

[2] Like with the previous translation, we should make sure that the other direction works as well.

[1] EdNote: cite

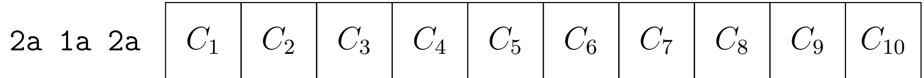
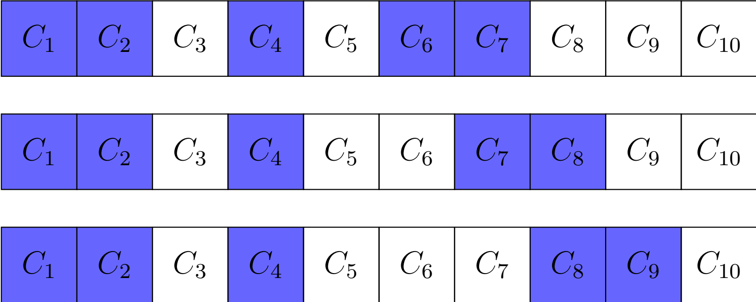



Figure 3: Running example for demonstrating the encodings. $C_i$ indicates if the $i$-th cell should be colored.

should color the cells. For a complete nonogram, we would have to encode all clues in the same way (but of course we have to be careful that we have a different color variable for each cell).

## B.1 Approach 1: Listing possible arrangements

This is the most obvious approach and you will find similar solutions when searching for "SAT-based nonogram solver". In this approach, you simply list all the possible arrangements and state that one of them must be true. The possible arrangements for the example clue would be:




...

We can easily express this as a DNF formula:

$$
\begin{aligned}
&(C_1 \land C_2 \land \neg C_3 \land C_4 \land \neg C_5 \land C_6 \land C_7 \land \neg C_8 \land \neg C_9 \land \neg C_{10}) \\
\lor\ &(C_1 \land C_2 \land \neg C_3 \land C_4 \land \neg C_5 \land \neg C_6 \land C_7 \land C_8 \land \neg C_9 \land \neg C_{10}) \\
\lor\ &(C_1 \land C_2 \land \neg C_3 \land C_4 \land \neg C_5 \land \neg C_6 \land \neg C_7 \land C_8 \land C_9 \land \neg C_{10}) \\
\lor\ &\ldots
\end{aligned}
$$

To convert the formula to CNF, it helps to introduce one helper variable for each arrangement (see also Appendix A.2).

## B.2 Approach 2: Use variables to denote each block start

In the example clue, we have three blocks, which we will refer to as $\alpha$, $\beta$ and $\gamma$. We will introduce variables $S_\xi^i$ for each $\xi \in \{\alpha, \beta, \gamma\}$ and $i \in \{1, \dots, 10\}$ to indicate the $\xi$ block starts at cell $i$.

For example, in the arrangement

| $C_1$ | $C_2$ | $C_3$ | $C_4$ | $C_5$ | $C_6$ | $C_7$ | $C_8$ | $C_9$ | $C_{10}$ |
|---|---|---|---|---|---|---|---|---|---|

the $\alpha$ block would start in cell 2, the $\beta$ block in cell 6 and the $\gamma$ block in cell 8. We would therefore represent the arrangement as $S_\alpha^2 \wedge S_\beta^6 \wedge S_\gamma^8$ (and all other $S_\xi^i$ would be false). Now we have to do two things:

1. link the block starts to cell colors and
2. make sure that we only get valid combinations of block starts.

We can link the block starts to cell colors with simple implications. For example, if the $\alpha$ block starts at position 1, then cells 1 and 2 must be colored (because, according to the clue, it is a block of length 2), i.e. $S_\alpha^1 \Rightarrow (C_1 \wedge C_2)$. Similarly, $S_\alpha^2 \Rightarrow (C_2 \wedge C_3)$ etc. While such constraints make sure that all cells that belong to a block are colored, we also need additional constraints to make sure that cells are *only* colored if they belong to a block.

Making sure we only get valid combinations of block starts is a bit trickier. First of all, we can use the exclusive or ($\oplus$) to make sure that each block has *exactly one* start:

$$S_\alpha^1 \oplus S_\alpha^2 \oplus S_\alpha^3 \oplus \dots$$

We can do the same for $\beta$ and $\gamma$. The last step is now to make sure that the blocks are appropriately spaced. There are different several ways to achieve this.

**Variant 1** Similarly to approach 1, we simply list all possible arrangements:

$$(S_\alpha^1 \wedge S_\beta^4 \wedge S_\gamma^6) \vee (S_\alpha^1 \wedge S_\beta^4 \wedge S_\gamma^7) \vee (S_\alpha^1 \wedge S_\beta^4 \wedge S_\gamma^8) \vee (S_\alpha^1 \wedge S_\beta^4 \wedge S_\gamma^9) \vee (S_\alpha^1 \wedge S_\beta^5 \wedge S_\gamma^7) \vee \dots$$

**Variant 2** We make sure that if one block starts at a particular position, the next block will start sufficiently late:

$$\begin{aligned} &(S_\alpha^1 \Rightarrow (S_\beta^4 \vee S_\beta^5 \vee \ldots)) \\ \wedge\ &(S_\alpha^2 \Rightarrow (S_\beta^5 \vee S_\beta^6 \vee \ldots)) \\ \wedge\ &\ldots \\ \wedge\ &(S_\beta^1 \Rightarrow (S_\gamma^3 \vee S_\gamma^4 \vee \ldots)) \\ \wedge\ &(S_\beta^2 \Rightarrow (S_\gamma^4 \vee S_\gamma^5 \vee \ldots)) \\ \wedge\ &\ldots \end{aligned}$$

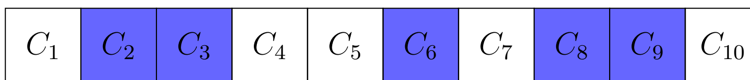
Note: It's a design choice whether you want to create impossible block starts like $S_\beta^1$ (the SAT solver would probably discard them very quickly).

**Variant 3** We make sure that if one block starts at a particular position, the next block will not start too early:

$$\begin{aligned} &(S_\alpha^1 \Rightarrow (\neg S_\beta^1 \wedge \neg S_\beta^2 \wedge \neg S_\beta^3)) \\ \wedge\ &(S_\alpha^2 \Rightarrow (\neg S_\beta^1 \wedge \neg S_\beta^2 \wedge \neg S_\beta^3 \wedge \neg S_\beta^4)) \\ \wedge\ &\ldots \\ \wedge\ &(S_\beta^1 \Rightarrow (\neg S_\gamma^1 \wedge \neg S_\gamma^2)) \\ \wedge\ &(S_\beta^2 \Rightarrow (\neg S_\gamma^1 \wedge \neg S_\gamma^2 \wedge \neg S_\gamma^3)) \\ \wedge\ &\ldots \end{aligned}$$

## B.3 Approach 3: Enclose blocks

Like in approach 2, we will name the three blocks in the clue $\alpha$, $\beta$ and $\gamma$. For each block $\xi \in \{\alpha, \beta, \gamma\}$ and each cell $i \in \{1, \ldots, 10\}$ we will now introduce two variables: $A_\xi^i$ and $B_\xi^i$. $A_\xi^i$ indicates that the block $\xi$ *starts after* cell $i$ and $B_\xi^i$ indicates that the block $\xi$ *ends before* cell $i$. For example, the arrangement



would be encoded as

$$\begin{aligned}&A_\alpha^1 \wedge \neg A_\alpha^2 \wedge \neg A_\alpha^3 \wedge \neg A_\alpha^4 \wedge \neg A_\alpha^5 \wedge \neg A_\alpha^6 \wedge \neg A_\alpha^7 \wedge \neg A_\alpha^8 \wedge \neg A_\alpha^9 \wedge \neg A_\alpha^{10}\\ \wedge\ &\neg B_\alpha^1 \wedge \neg B_\alpha^2 \wedge \neg B_\alpha^3 \wedge B_\alpha^4 \wedge B_\alpha^5 \wedge B_\alpha^6 \wedge B_\alpha^7 \wedge B_\alpha^8 \wedge B_\alpha^9 \wedge B_\alpha^{10}\\ \wedge\ &A_\beta^1 \wedge A_\beta^2 \wedge A_\beta^3 \neg A_\beta^4 \wedge A_\beta^5 \wedge \neg A_\beta^6 \wedge \neg A_\beta^7 \wedge \neg A_\beta^8 \wedge \neg A_\beta^9 \wedge \neg A_\beta^{10}\\ \wedge\ &\ldots\end{aligned}$$

Now, cell $i$ should be colored if and only if it belongs to one of the blocks, i.e. if one of the blocks does not start after $i$ and does not end before $i$. So, in our example, we would e.g. have for cell 5

$$C_5 \Leftrightarrow ((\neg A_\alpha^5 \wedge \neg B_\alpha^5) \vee (\neg A_\beta^5 \wedge \neg B_\beta^5) \vee (\neg A_\gamma^5 \wedge \neg B_\gamma^5))$$

We also put some basic constraints on our $A$ and $B$ variables. If a block starts after cell $i$, it also starts after cell $i-1$, i.e.

$$(A_\alpha^2 \Rightarrow A_\alpha^1) \wedge (A_\alpha^3 \Rightarrow A_\alpha^2) \wedge (A_\alpha^4 \Rightarrow A_\alpha^3) \wedge \ldots \wedge (A_\beta^2 \Rightarrow A_\beta^1) \wedge (A_\beta^3 \Rightarrow A_\beta^2) \wedge (A_\beta^4 \Rightarrow A_\beta^3) \wedge \ldots$$

We can treat the $B$ variables analogously. As "$\Rightarrow$" is transitive, the above formula also implies e.g. $A_\alpha^4 \Rightarrow A_\alpha^2$.

At last, we also have to make sure that the blocks have the right length. For example, the $\alpha$ block should have length 2 according to the clue. This means that if the block does not start after $i$ (i.e. it starts at $i$ or before $i$), then it should end before $i+2$. Concretely, that gives us

$$((\neg A_\alpha^1) \Rightarrow B_\alpha^3) \wedge ((\neg A_\alpha^2) \Rightarrow B_\alpha^4) \wedge ((\neg A_\alpha^3) \Rightarrow B_\alpha^5) \wedge \ldots$$

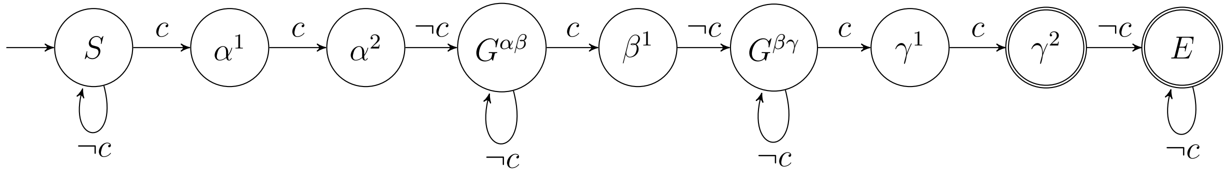
## B.4 Approach 4: Make an automaton

In this approach, we will effectively create something like a finite automaton[3]. The input is a sequence of cell colors and the automaton will only accept sequences that match the clue. Each state intuitively describes the role of the previously entered cell. For the example clue, we would have the following states (again calling the 3 blocks $\alpha$, $\beta$ and $\gamma$):

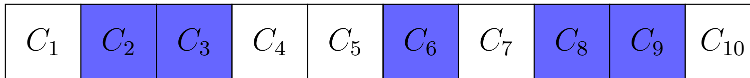
[3]For simplicity, we do not exactly follow the traditional definition of a deterministic finite automaton. In particular, the transition function is not total.

| State | Role of previously entered cell color |
|---|---|
| $S$ | Cell before the first ($\alpha$) block |
| $\alpha^1$ | First cell of the $\alpha$ block |
| $\alpha^2$ | Second cell of the $\alpha$ block |
| $G^{\alpha\beta}$ | Cell in the gap between the $\alpha$ and the $\beta$ block. |
| $\beta^1$ | First cell of the $\beta$ block |
| $G^{\beta\gamma}$ | Cell in the gap between the $\beta$ and the $\gamma$ block. |
| $\gamma^1$ | First cell of the $\gamma$ block |
| $\gamma^2$ | Second cell of the $\gamma$ block |
| $E$ | Cell after the $\gamma$ block |

Writing $c$ for a colored cell and $\neg c$ for an uncolored cell, the full automaton is:




The automaton starts in state $S$ and accepts the input if it ends up in state $\gamma^2$ or $E$. It would, for example, accept the sequence $\neg c, c, c, \neg c, \neg c, c, \neg c, c, c, \neg c$, which corresponds to the arrangement




The state sequence for that input is $S, \alpha^1, \alpha^2, G^{\alpha\beta}, G^{\alpha\beta}, \beta^1, G^{\beta\gamma}, \gamma^1, \gamma^2, E$. An input starting with $c, c, c$ (which is not allowed according to the clue) would get rejected because a $c$ input is not allowed in state $\alpha^2$.

To represent the automaton in propositional logic, we will introduce variables $\Sigma_i$ indicating that we are in state $\Sigma$ after processing the input until (and including) cell $i$. For example, if the first cell is colored ($C_1$ is true), then we will be in state $\alpha^1$ after processing it. If it is not colored, on the other hand, we will remain in state $S$. That gives us the following formulae:

$$C_1 \Rightarrow \alpha_1^1$$
$$\neg C_1 \Rightarrow S_1$$

If we are now in state $S$ (i.e. $S_1$ is true), then, depending on the next cell color, we will either

end up in state $S$ or $\alpha^1$:

$$(S_1 \land \neg C_2) \Rightarrow S_2$$
$$(S_1 \land C_2) \Rightarrow \alpha_2^1$$

But if we were in state $\alpha^1$ (i.e. $\alpha_1^1$ is true), then the next cell has to be colored ($C_2$ must be true) and we will end up in state $\alpha^2$:

$$\alpha_1^1 \Rightarrow C_2 \land \alpha_2^2$$

Here are the formulae for cell 3:

$$(S_2 \land \neg C_3) \Rightarrow S_3$$
$$(S_2 \land C_3) \Rightarrow \alpha_3^1$$
$$\alpha_2^1 \Rightarrow (C_3 \land \alpha_3^2)$$
$$\alpha_2^2 \Rightarrow (\neg C_3 \land G_3^{\alpha\beta})$$

We can do the same for the remaining cells. However, we must ensure that, after processing the last cell (cell 10), we are in state $\gamma^2$ or $E$, so we should add one more constraint:

$$\gamma_{10}^2 \lor E_{10}$$

**Note:** If you also support impossible states in your encoding (e.g. $\gamma_1^1$), you might have to add additional constraints to make sure that the automaton will only be in one state at each cell. Otherwise, your automaton might end up in two states simultaneously for a cell (one valid and one impossible), and then continue with the impossible state to reach an impossible terminal state. A better approach would be to exclude any impossible states in the first place.