Theory Of Computation

FASHION STORE PROBLEM

May 30, 2023

Andrea Prato

Giacomo Solaro

Samuel Corecco

Xiana Carrera Alonso

Università della Svizzera italiana

Contents

1 Introduction

This report will describe the implementation of a solver for the Fashion Store Problem based on the SAT solver Z3. The main goals of the project were to outline the rules of the problem, formalize them, and design a proper encoder and a decoder to translate the input into a format recognizable by Z3 and then translate back its output into a format understandable by the user.

In order to provide an accessible and straightforward interface, we decided to develop the project as a web page.

2 The problem

The Fashion Store Problem states as follows:

> *Given a set of colored garments find a combination to dress your mannequin.*
>
> ***Constraints:*** *Some colors or garments don't go together, too many colors might be extravagant, too few colors are dull...*
>
> ***Task:*** *Write an encoder for Fashion Store Problem, that takes as input a text file containing a list of pairs, ⟨ garment, color ⟩, and encodes the problem into SAT. If the problem is unsatisfiable, output UNSATISFIABLE. If it is satisfiable, output a full set of garments satisfying all constraints.*

2.1 Domain definition

The problem statement only provided a basic guideline, so its specific input, constraints, variable domain, etc. were up to us to decide.

We started by defining the sets of garments and colors that would be available to the user. We chose eight garments and seven colors, listed in Table 1 and Table 2, respectively.

Garment
T-shirt
Pants
Hat
Shoes
Socks
Dress
Skirt
Sandals

Table 1: Garments

Color
Red
Blue
Violet
White
Green
Black
Pink

Table 2: Colors

Afterwards, we defined the constraints that the outfits would need to satisfy to be accepted. We started by writing the conditions/clauses in English, to then translate them into propositional logic and finally into disjunctions, so that we could write the SAT formula in CNF (Conjunctive Normal Form).

The choice of the clauses was a combination of some standard rules (e.g., it is not feasible to wear a skirt and pants at the same time) and more peculiar ideas meant to increase the complexity of the problem.

Although every formula in propositional logic can be converted into CNF ([3]), this is sometimes cumbersome and is not actually required by the Z3 solver (see later section on implementation), but we have included the necessary clause equivalences here for completeness. Note that after the mathematical simplification, some complex clauses end up turning into the conjunction of several smaller clauses.

- **Clauses regarding garments:**

  1. Certain pairs of garments cannot be chosen together. They are g_1 = (sandals, socks), g_2 = (skirt, pants), g_3 = (dress, t-shirt) and g_4 = (sandals, shoes). In propositional logic, we can express this as:

  $$\neg\,(g_i\,[0]\;\&\;g_i\,[1]) \quad\iff\quad \neg\,g_i\,[0] \vee \neg\,g_i\,[1]$$

  for $i \in \{1, 2, 3\}$. The equivalence results from the direct application of De Morgan's law ([1]).

  2. Either shoes or sandals must be worn.

  $$\text{shoes} \vee \text{sandals}$$

  3. Either a dress, a T-shirt with pants or a T-shirt with a skirt must be included.

  $$\begin{aligned}
  &\text{dress} \vee (\text{T-shirt} \wedge \text{pants}) \vee (\text{T-shirt} \wedge \text{skirt}) \iff \\
  &[\text{dress} \vee (\text{T-shirt} \wedge \text{pants}) \vee \text{T-shirt}] \wedge [\text{dress} \vee (\text{T-shirt} \wedge \text{pants}) \vee \text{skirt}] \iff \\
  &[\text{dress} \vee \text{T-shirt} \vee (\text{T-shirt} \wedge \text{pants})] \wedge [\text{dress} \vee \text{skirt} \vee (\text{T-shirt} \wedge \text{pants})] \iff \\
  &[(\text{dress} \vee \text{T-shirt} \vee \text{T-shirt}) \wedge (\text{dress} \vee \text{T-shirt} \vee \text{pants})] \wedge \\
  &\qquad \wedge [(\text{dress} \vee \text{skirt} \vee \text{T-shirt}) \wedge (\text{dress} \vee \text{skirt} \vee \text{pants})] \iff \\
  &(\text{dress} \vee \text{T-shirt}) \wedge (\text{dress} \vee \text{T-shirt} \vee \text{pants}) \wedge \\
  &\qquad \wedge (\text{dress} \vee \text{skirt} \vee \text{T-shirt}) \wedge (\text{dress} \vee \text{skirt} \vee \text{pants})
  \end{aligned}$$

  The first and third equivalences just use the fact that disjunctions are distributive over conjunctions. The second equivalence uses the commutativity of disjunctions, and the fourth one eliminates parenthesis and uses the idempotence of the disjunction operation. The final result is the conjunction of four irreducible clauses.

  4. If shoes are worn, then socks must be worn as well:

  $$(\text{shoes} \implies \text{socks}) \iff (\neg\,\text{shoes} \vee \text{socks})$$

  where here he have just expressed and implication using negation and disjunction only.

  5. If a dress is worn, then a hat must be worn as well:

  $$(\text{dress} \implies \text{hat}) \iff (\neg\,\text{dress} \vee \text{hat})$$

- **Clauses regarding colors**:

  1. Certain pairs of colors cannot be present together. They are c_1 = (blue, green), c_2 = (pink, green), c_3 = (green, red), c_4 = (blue, red) and c_5 = (pink, red). In propositional logic,

  $$\neg\,(c_i\,[0]\;\&\;c_i\,[1]) \quad\iff\quad \neg\,c_i\,[0] \vee \neg\,c_i\,[1]$$

  for $i \in \{1, 2, 3, 4, 5\}$.

  2. Mono-color outfits are not allowed. That is, at least two different colors must be present.

  $$\bigvee_{1 \le i < j \le 7} \text{color}_i \wedge \text{color}_j$$

  where *color* can be any of the options defined in Table 2.
  To transform this clause into a valid form, note that:

  $$(a \wedge b) \vee (c \wedge d) \iff ((a \wedge b) \vee c) \wedge ((a \wedge b) \vee d) \iff (a \vee c) \wedge (b \vee c) \wedge (a \vee d) \wedge (b \vee d)$$

  By recursively applying this property, we can simplify the clause into the conjunction of disjunctions.

3. An outfit cannot have more than 3 colors. Equivalently, since we have 7 colors in total, at least 4 of them must not be selected.

$$\bigvee_{1 \leq i_1 < i_2 < i_3 < i_4 \leq 7} \neg \, \text{color}_{i_1} \wedge \neg \, \text{color}_{i_2} \wedge \neg \, \text{color}_{i_3} \wedge \neg \, \text{color}_{i_4}$$

The transformation of this clause would be done in the same way as in the previous case.

- **Clauses regarding combinations of garments and colors**:

1. Violet hats cannot be worn together with pink dresses, and pink hats cannot be worn together with violet dresses.

$$\neg \, (\text{violet} \wedge \text{dress} \wedge \text{pink} \wedge \text{hat}) \iff \neg \, \text{violet} \vee \neg \, \text{dress} \vee \neg \, \text{pink} \vee \neg \, \text{hat}$$

by applying De Morgan's law.

The final CNF SAT formula, φ, is just the conjunction of all of the resulting clauses with the form of disjunctions of literals (variables or negated variables).

2.2 Objective

If we wished to know if there is a possible solution to the problem defined above, we could introduce it into the Z3 Sat Solver, which would search for an interpretation I such that $I \models \varphi$, i.e., such that the well-formed formula φ evaluated to true under I, which assigns to every variable (garments and colors) exactly one truth value.

Additionally, we have another objective: to determine if an interpretation exists after the user has made a selection of clothes. This selection would be expressed as two new clauses: one for imposing the inclusion of every choice the user has made, and another one for imposing the negation of every item and color the user has not selected. If the output of the program is "satisfiable", that means that it is a valid outfit. If it is "unsatisfiable", it is not. This does not imply that no additional clothes can be added, nor that no clothes can be removed: it simply should be understood as recognising a valid solution.

3 Implementation

3.1 Encoding

The encoder bears the task of converting the input into a format that is acceptable by the solver. Said input is provided as a text file, to facilitate testing and the search for the performance limits. Therefore, the user input through the GUI is first converted to a text file before being processed.

When reading the file, we first check the format:

1. Each line must have exactly two parts, separated by a comma (',').
2. The first part must be one of the garments shown in Table 1.
3. Garments cannot be repeated.
4. The second part must be one of the colors shown in Table 2.
5. All text must be in lowercase.

All of these checks must be satisfied. If not, the program signals an error and stops prematurely. Note that the constraint "garments cannot be repeated" cannot be expressed directly as a clause in φ, and that is the reason why we validate it beforehand.

Afterwards, we initialize a solver and convert each of the input garments and colors into boolean variables:

```
# Create a Z3 solver
solver = Solver()

# Dictionaries for the input variables
garment_vars = {}
color_vars = {}

# Transform the user garments and colors into Z3 variables
for garment in garments:
    garment_vars[garment] = Bool(garment)

for color in colors:
    if color not in color_vars:
        color_vars[color] = Bool(color)
```

Listing 1: Variable initialization

Then, we add the "fixed constraints" to the solver, that is, the rules that we have previously defined and that constitute the core of the problem, invariant to each execution. Following the documentation available in [2], we were able to simplify the code by using a wider range of propositional logic operators than simply conjunctions, disjunctions and negations, with implications, operators such as "AtMost", etc. Additionally, Z3 does not require input of the SAT formula, φ, to be in CNF. This meant that the constraints could be introduced in a higher-level, more easily readable way than with the developed formulas previously stated.

```
# Constraint: Either a dress or (a Tshirt and pants) or (a Tshirt and skirt)
solver.add(Or(Clothes['dress'], And(Clothes['tshirt'], Clothes['pants']),
           And(Clothes['tshirt'], Clothes['skirt'])))

# Constraint: Shoes imply socks
solver.add(Implies(Clothes['shoes'], Clothes['socks']))

# Constraint: No more than 3 colors
solver.add(AtMost(*[Colors[color] for color in Colors], 3))
```

Listing 2: Some examples of constraints definitions. Clothes and Colors are dictionaries that map the input to its corresponding boolean variable.

After that, we have two possibilities:

1. Provide a possible solution to the problem, i.e., a combination of garments and colors that satisfies the rules we have imposed.
2. Check if the outfit read from the text file is valid.

The first option is straightforward thanks to the Z3 interface. For the second option, we first need to convert the user input into constraints by adding two new clauses that impose to include all chosen garments and colors, and to exclude all those that were not chosen:

```
def add_input_constraints(solver, garment_vars, color_vars):
    clothes_copy = Clothes.copy()
    colors_copy = Colors.copy()

    chosen_garments = []
    chosen_colors = []

    for garment in garment_vars:
        chosen_garments.append(garment_vars[garment])
        clothes_copy.pop(garment)
```

```
    for color in color_vars:
        chosen_colors.append(color_vars[color])
        colors_copy.pop(color)

    # All the garments chosen by the user must be present in the solution
    solver.add(And(*chosen_garments))
    # All the colors chosen by the user must be present in the solution
    solver.add(And(*chosen_colors))

    # None of the garments not chosen by the user can be present in the
    solution
    solver.add(And(*[Not(clothes_copy[garment]) for garment in clothes_copy
    ]))
    # None of the colors not chosen by the user can be present in the
    solution
    solver.add(And(*[Not(colors_copy[color]) for color in colors_copy]))
```

Listing 3: Adding constraints from the user input

3.2 Decoding

Finally, we check if the problem is satisfiable or not. If it is, that means that the user's choice is compliant with our rules, and we provide an affirmative answer.

On the other hand, if no assignment of variables is possible, then some part of the choice must be a contradiction with our rules. Thus the output is negative.

The role of the decoder when checking a user selection is trivial (see figure 1), since the solution is exactly the outfit that the user had provided. Therefore, in this case the output is binary (valid/not valid).

However, if the program was used to provide a possible solution for the (fixed) problem constraints (figure 2), decoding also involves converting the result given by Z3 into a readable format, essentially by outputing those garments and colors that have been assigned a value of 1 (i.e., that should be selected).

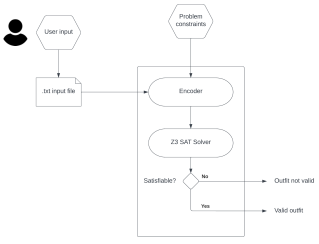

Figure 1: Architecture in the "user selection validation" mode.

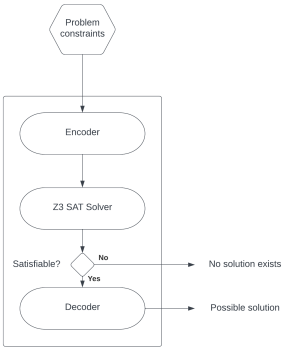

Figure 2: Architecture in the "general solution search" mode.

3.3 Testing

To verify that the implementation of the solver was compliant with our expectations, we wrote 10 test cases that check the output on a series of text files. The files are autogenerated by the tests and examine each of the constraints, so that we can conclude that if any one of them fails, the output is "unsatisfiable". There are also positive (satisfiable) cases, from

which we can deduce that there are possible solutions to the problem. Testing can be done by executing the *test.py* file.

```
    def test_shoes_or_sandals(self):
    with open('tests/shoes_or_sandals.txt', 'w') as f:
        f.write('tshirt, white\n')
        f.write('pants, black\n')
        f.write('hat, black\n')
        f.write('socks, white\n')
        f.close()
    self.assertFalse(main.run("tests/shoes_or_sandals.txt"))
```

Listing 4: Example of test case

4 Efficiency

To test the limits of the solver, we progressively added new garments and colors (i.e., Boolean variables) in a series of tests over a default input (*test_list.txt*). Since none of these extra variables appears in the input file, they are added to the solver as part of the clauses that impose that the colors and garments not chosen by the user are assigned False in the solution (see Listing 3). The fixed clauses of the problem were not changed. For each execution, we collected the statistics provided by Z3 and measured the total time required. The relevant file for this section is *performance.py*.

At first, we were unsure if this would be an adequate approach. Since we were not changing the fixed clauses, we thought that the solver might have optimizations that caused the new variables to be irrelevant and ultimately have no impact on the time. However, this proved to be wrong: the number of variables had direct consequences on the difficulty.

In table 3 we can see a exponential tendency on the time taken for each execution, that is even more clear when the results are plotted (figure 3). n stands for the number of extra garments and extra colors added ($2n$ variables in total). Note that the focus should be on the overall tendency and not on the instances with a small value of n, which are not indicative of the complexity of the problem. Additionally, all of the times are quite reasonable, even for large inputs, because of the fact that we are not increasing the number of clauses (we are just changing the number of literals in two of them), which restrains the difficulty of this specific problem from growing too quickly.

We tried one final attempt with $n = 1000000$, which took 1 hour and 51 minutes, showing how prohibitive large instances can be.

n	Memory (MB)	Total time (s)
1	20.73	0.0179
10	20.68	0.017
100	20.73	0.025
1000	21.23	0.101
5000	31.36	0.6169
10000	50.69	1.1709
50000	171.08	14.5042
70000	314.15	26.0611
90000	320.88	41.1998
100000	329.21	93.8406

Table 3: Performance results.

It is also interesting to note the increase in the necessary amount of memory. In contrast to the time complexity, we can observe a linear tendency, with a main factor slightly lower than 1.

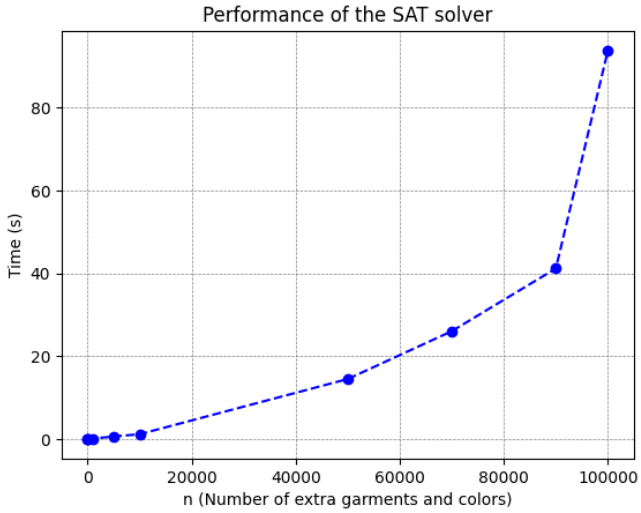

Figure 3: Time of the executions with respect to size of the input.

5 Application

5.1 Running Instructions

As running our application may result a bit complex for new users, we will briefly explain how to start it here.

Firstly, Google CORS policies must be disabled, because they prevent communication between the local host and an element of the file system. This can be achieved, in Linux, by running the following command:

```
$ google-chrome --disable-web-security
```

For macOS the command is the following:

```
$ open -n -a /Applications/Google\ Chrome.app/Contents/MacOS/Google\
Chrome --args --user-data-dir="/tmp/chrome_dev_test" --disable-web-
security
```

This will generate a Google window in which CORS policies will be disabled.

After disabling the CORS policies, the server running the backend must be activated. This can be done through the command:

```
$ python3 server.py
```

Finally, the application can be entered by entering the page URL in the Google window which popped up after running the first command. Once entered the URL the GUI of the application will appear in front of the user.

To run the test cases, we recommend to user the -b option, as in `test.py -b`, to suppress the output from the main program.

5.2 Graphic User Interface

The graphical user interface (GUI) is designed to be user-friendly and intuitive.

Figure 4: Website GUI

5.2.1 Character

The character on the left side of our web interface provides a real-time reference of the combination of clothes selected by the user. Figure 4 updates dynamically each time a piece of clothing is chosen, offering an immediate visual representation of the choices made. Although it is not possible to interact directly with this mannequin, its presence plays a key role: it makes the decision-making process more intuitive, allowing the user to appreciate the concrete impact of their decisions. In this way, the interaction with the website becomes a more engaging and intuitive experience.

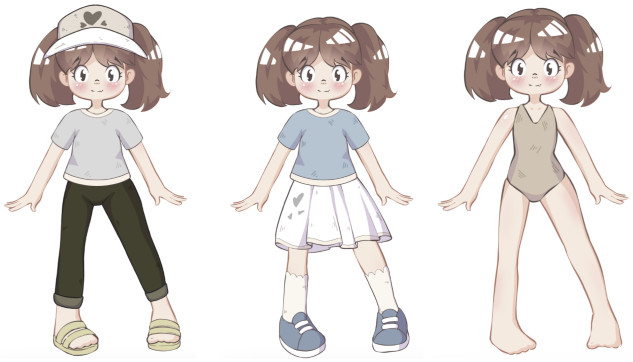

Figure 5: example of character's variant

5.2.2 Inventory and Shop

Positioned on the right, the user can find the inventory. This features a selection panel for choosing the type of clothing, which includes Hats, Shirts, Suits, Skirts, Trousers, Shoes, Sandals, and Socks. Upon selecting a category, an array of items of that specific type appears in all available colors. This structure allows for smooth and simple navigation among all clothing categories and their variants.

To select an item, simply click on it, and it will be automatically added to the mannequin situated on the left. However, from this section, it is not possible to manually remove the chosen clothing category; it is, however, possible to replace it with another item from the same category, but of a different version.

On the other hand, the shop serves the purpose of listing all the items previously selected from the inventory, thus providing an organized list of the selected garments. From this section, if desired, it is possible to remove one of the chosen items, without necessarily having to replace it with another.

Figure 6: inventory

Figure 7: Shop

5.2.3 Validation

In our user interface, once the desired outfit has been selected, it's possible to verify if the chosen combination adheres to the constraints defined in the backend. To do this, simply click on the dedicated verification button: at this point, the frontend sends the data to the backend for validation, receiving a response indicating whether the outfit meets the requirements or not.

If the outfit is deemed suitable, everything proceeds smoothly. A confirmation message will appear, reassuring us about the correctness of the current outfit, accompanied by a summary of the choices made.

If, on the other hand, the outfit does not meet the constraints, an error message will appear to signal the inconsistency. In addition to this message, the currently selected outfit will be automatically cleared, returning the situation to the initial state, and allowing the user to restart the selection process from scratch.

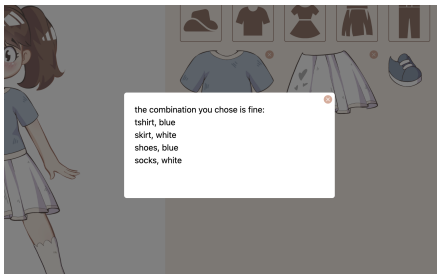

Figure 8: outfit is good

Figure 9: outfit is wrong

6 Problems encountered

We faced several challenges with the loading of the images due to the inclusion of 42 cropped images and 42 A4-sized images, resulting in a total of 84 high-resolution images (2480x3508). One significant issue was the need to replace the images whenever the clothing type on the website changed. This would have required deleting the old images and uploading the new ones, which would have been computationally expensive to perform repeatedly.

Consequently, we encountered two main problems. The first challenge arose from our initial choice of using React for frontend development. Importing each image individually, as

required by React, would have made the development process cumbersome and inefficient, given the substantial number of images involved.

The second problem was closely related to the weight of the files: each change in the clothing displayed required a new loading of the images, a process that was extremely heavy.

To solve these issues, we chose a different approach. We decided to load all the elements only once, setting their style to "display: none". In this way, the images would not have to be reloaded from scratch every time, making the process more efficient and less resource-intensive.

7 Alternatives considered

For a while, we considered implementing an option to let the user choose custom constraints. However, when thinking about how to present it, we realized that this feature had the risk of running into contradictions with our "hidden" rules without the user realizing it, which could lead to frustration. One possible solution would be to expose said rules, but since we believe that figuring them out is part of the fun and this idea did not add any additional complexity to the solver, we ultimately decided to scrap it.

8 Work distribution

The workload was distributed as follows:

- Andrea: frontend, report, presentation.
- Giacomo: backend, report writing, presentation.
- Samuel: frontend, report, presentation.
- Xiana: backend, report writing, presentation.

9 Conclusion

This assignment allowed us to get a first-hand experience with a SAT Solver. Due to its room for creativity, it also meant that we could work with more complex SAT clauses and figure out how to simplify them and implement them in Python. We consider that our understanding of the complexity of SAT has improved, and also of all of its potential applications.

Bibliography

[1] Isaac Computer Science. National Centre for Computing Education. *De Morgan's laws*. URL: `https://isaaccomputerscience.org/concepts/sys_bool_de_morgans?examBoard=all&stage=all`. (accessed: 22.05.2023).

[2] Microsoft. Archive of files in old website. *Z3py tutorial*. URL: `https://ericpony.github.io/z3py-tutorial/guide-examples.htm`. (accessed: 28.05.2023).

[3] Applied Logic for Computer Science Course Zhen Xie Y. University of Western Ontario. *Propositional Logic: Conjunctive Normal Form Disjunctive Normal Form*. URL: `https://www.csd.uwo.ca/~mmorenom/cs2209_moreno/slide/lec8-9-NF.pdf`. (accessed: 22.05.2023).