Theory of Computation
Assignment 6: SAT Encoding

Brites Marto Andrea Le Thuong
Rodolfo Masera Tommaso Taillefert Stefano

1 Installation and Instructions

We kindly ask you to refer to the `README.md` file that we will have attached to the assignment submission in order to install the needed dependencies and run our application properly.
For completeness, we will also put the instructions directly in this document.
First of all, you will find a text file named `requirements.txt` that contains the Python dependencies to be installed in order to run the application. You can simply run in your console:

```
pip install --no-cache-dir -r requirements.txt
```

Then, if you'd like to run the program from CLI, you can simply do as follows:

```
python main.py input_file
```

where "`input_file`" is a text file with the pairs of garments and colours $\langle g, c \rangle$. See under "`examples/`" or below for proper formatting.
If you instead would prefer to run the web server and the web application we developed, refer to these instructions:

```
cd app
flask run
```

Head to `localhost:5000` from your browser to access the interface. If you'd like to run it with Docker, you can run `docker-compose up` from your terminal instead.

For an example of what an input file should look like, refer to what follows here:
A text file in the following format where the first line `garment, color` should **NOT** be changed and should **ALWAYS** be present to then follow with the pairs you would like to feed to the script. Here is an example:

```
garment, color
shirt, black
shorts, white
jacket, blue
```

2 Problem Design and Interpretation

First and foremost this problem is very loosely defined, which means that it was mostly up to us to come up with constraints and with an appropriate problem size. Our input is a pair of garments and colours $\langle g, c \rangle$, where $g \in G$ and $c \in C$; G and C are sets that include garments and colours respectively, both of size 10, and are defined as follows:

$$G = \{\text{pants, shirt, hat, jacket, sweater, gloves, shoes, tie, scarf, shorts}\}$$
$$S = \{\text{red, yellow, orange, green, blue, purple, brown, pink, white, black}\}$$

We chose a small size for the sake of simplicity of the project and for a more realistic aspect. We will go over this part in more detail in **section 4**.
Given these two sets, we devised constraints over them that will always be added. They are hardcoded as they specify, for instance, which garments (or colours) should or should not go together. We define these constraints with the following boolean expressions:

Boolean Constraints
$\neg$(yellow $\wedge$ white)
$\neg$(blue $\wedge$ purple)
$\neg$(blue $\wedge$ black)
$\neg$(red $\wedge$ green)
$\neg$(red $\wedge$ orange)
$\neg$(green $\wedge$ pink)
$\neg$(green $\wedge$ orange)
$\neg$(pants $\wedge$ shorts)
$\neg$(shorts $\wedge$ jacket)
scarf $\rightarrow$ jacket
gloves $\rightarrow$ jacket
tie $\rightarrow$ shirt

Finally, we go over the given input file and we determine the final constraints. If one of the garments is missing, we make sure that they are put in a **NOT** boolean operator to take their absence into account; otherwise, if present, we simply add to our constraints list an **OR** chain of **AND**s over the pairs comprising a garment g and its color c. So basically, over an input $\langle g_1, c_1 \rangle, \langle g_2, c_2 \rangle$, we would obtain the CNF $(g_1 \wedge c_1) \vee (g_2 \wedge c_2) \vee \neg g_3 ... \vee \neg g_n$, plus all the constraints from the table above.

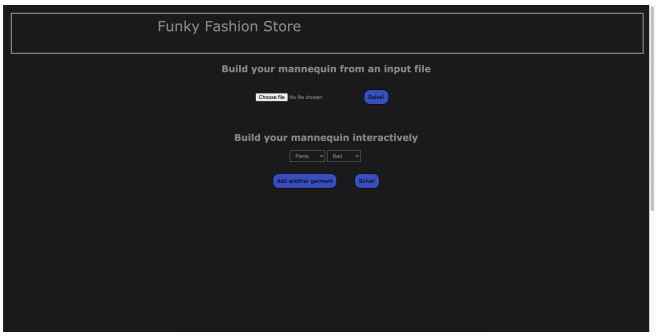
3 Implementation

3.1 SAT Solver

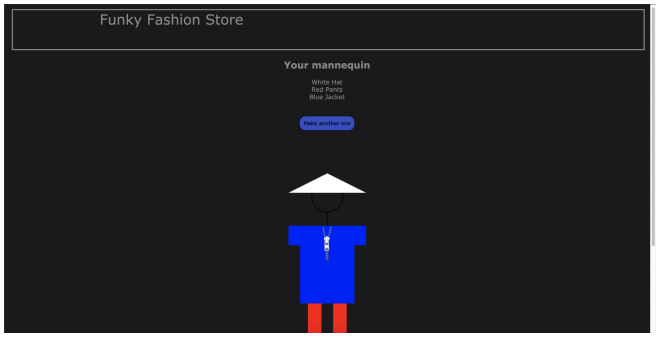
The main core of the project is the solver program (`main.py`), written entirely in Python 3. We used the `z3-solver` library (`https://github.com/Z3Prover/z3`) to get most of the job done since it provided us with the functionalities we needed and was very straightforward to use. We also used `pandas` to read the input file. No, we did not ask a bunch of fluffy animals to read out the text, it's a Python library. Come on.
Our program first puts the input pairs of garments and colors into a set, to filter out duplicates. Then, we build the constraints list as detailed in section 2 and we check whether the model is satisfiable or not. If it's the case, we print out the solution.

3.2 User Interface

We needed to quickly develop a solid interface; the perfect candidate for this, given our skills, was a webpage. Therefore, given that the rest of the code was already in Python, we resorted to `Flask`, which is a simple yet powerful library to make web applications.
The script (`app/app.py`) simply renders an HTML page with two inputs to compose the mannequin and a canvas to represent the solution, which is obtained by calling our solver with the given input data.



(a) The homepage



(b) A solved mannequin

3.3 Deployment

Since we built a web interface, we thought it would be a good idea to deploy our project as a fully functional website. For this reason, we purchased the domain `bestsatsolver.xyz`, packed our app in a Docker container and put it on a small server of ours. The whole process was pretty straightforward, given that our project has a very simple structure and doesn't depend on other services like databases or external APIs.

4 Issues Encountered

Other than the typical issues that may arise when programming (i.e. bugs, regretting some code design choices, ...), the main issue that we encountered ended up being caused by the fact that the problem assigned to us was loosely defined. This meant that we had more freedom to choose how to develop it but that we also had to make sure to keep it complete while not overcomplicating it.

As we described in **section 2**, we decided to keep the size of the problem relatively small by limiting the possible colors and garments to a set of ten elements each. Along with the fact that, in reality, oftentimes we would not expect to have to dress a mannequin with thousands of garments and colours, we also thought that it would be easier to define our own sets and constraints over them rather than to allow variable input sizes.

The main issue that would have come with a variable input size where you would allow the input to decide how many garments and colours would be available (along with the subsequent possible pairs) would have been to define the constraints over the sets of colours and garments.

For instance, if we take our own sets, we have ten garments and ten colours and we have added constraints with the knowledge of those garments and colours. If, instead, we had x garments and y colours where $x, y \in \mathbb{N}$, we then would first of all have $x \cdot y$ possible pairs and, furthermore, defining the constraints over them would be incredibly tough and we might have to take a different - perhaps randomized - approach.

Therefore, given the fact that a variable input size with respect to the number of garments and colours would be tougher to work with, we have delimited the problem to two given sets of size ten each where we can clearly define our constraints.

Another problem we discovered while testing edge cases is the following: if the input contains two same garments but of different colors, sometimes the model will allow both and the mannequin will end up e.g. with two pairs of pants on. This stems from the fact that it's very hard to dynamically put an exclusivity constraint to rule out the phenomenon; technically speaking, the `Xor()` method doesn't support varargs, so we would have to combine each garment in pairs and check all combinations one by one (madness!). Simply filtering the output set felt like cheating with regard to what the model is actually computing, so we solved it by simply considering that the model *offers* you multiple choices when possible. It's not a bug, it's a feature!