Project – I
Mass Customization

CSCI 2951-O: Foundations of Prescriptive Analytics

Preliminary Due: February 24rd, 2017
Final Due Date: March 3rd, 2017

1 Problem Statement

The ability to provide customized products has become ever more important across a wide range of industries. End-users increasingly demand unique products that are tailored for their specific needs. To that end, Mass Customization tackles the challenge of producing goods and services to meet individual customer's needs at minimum cost.

In exciting news, Electronic Cars Consortium (ECC) has sealed a deal with the Italian luxury car manufacturer, Ferrari, to offer a new generation of highly customizable hybrid sports cars. As can be seen in Figure 1, compilation of correct customization data is a complex task in the automotive industry. To continue Ferrari's beautiful, stylish, and exquisite offerings, ECC is now tasked with delivering an intelligent method for detecting whether various configurations are feasible or not. Faced with such a challenge, ECC has no better option than turning to CSCI 2951-O elites!

2 Input

Based on its customer knowledge base, ECC compiled a number of customizations into a set of instances. These instances are stored in the standard Conjunctive Normal Form (CNF). The CNF format is used to define a Boolean formula over Boolean variables involving negation (NOT $\neg$), conjunction (AND $\wedge$) and disjunction (OR $\vee$) operations.

A CNF formula consists of *clauses* joined by AND. In turn, each clause consist of Boolean *literals* joined by OR. Each literal either corresponds to a Boolean variable, referred to as positive literal, or to its negation, referred to as negative literal. For example, F = $(x2 \vee \neg x1) \wedge (x3 \vee \neg x2 \vee x1)$ is a formula in conjunctive normal form.

2.1 CNF File Format

A CNF file might have comment lines which start with the letter "c". This is followed by the problem line which starts with the letter "p", followed by the problem type ("cnf"),

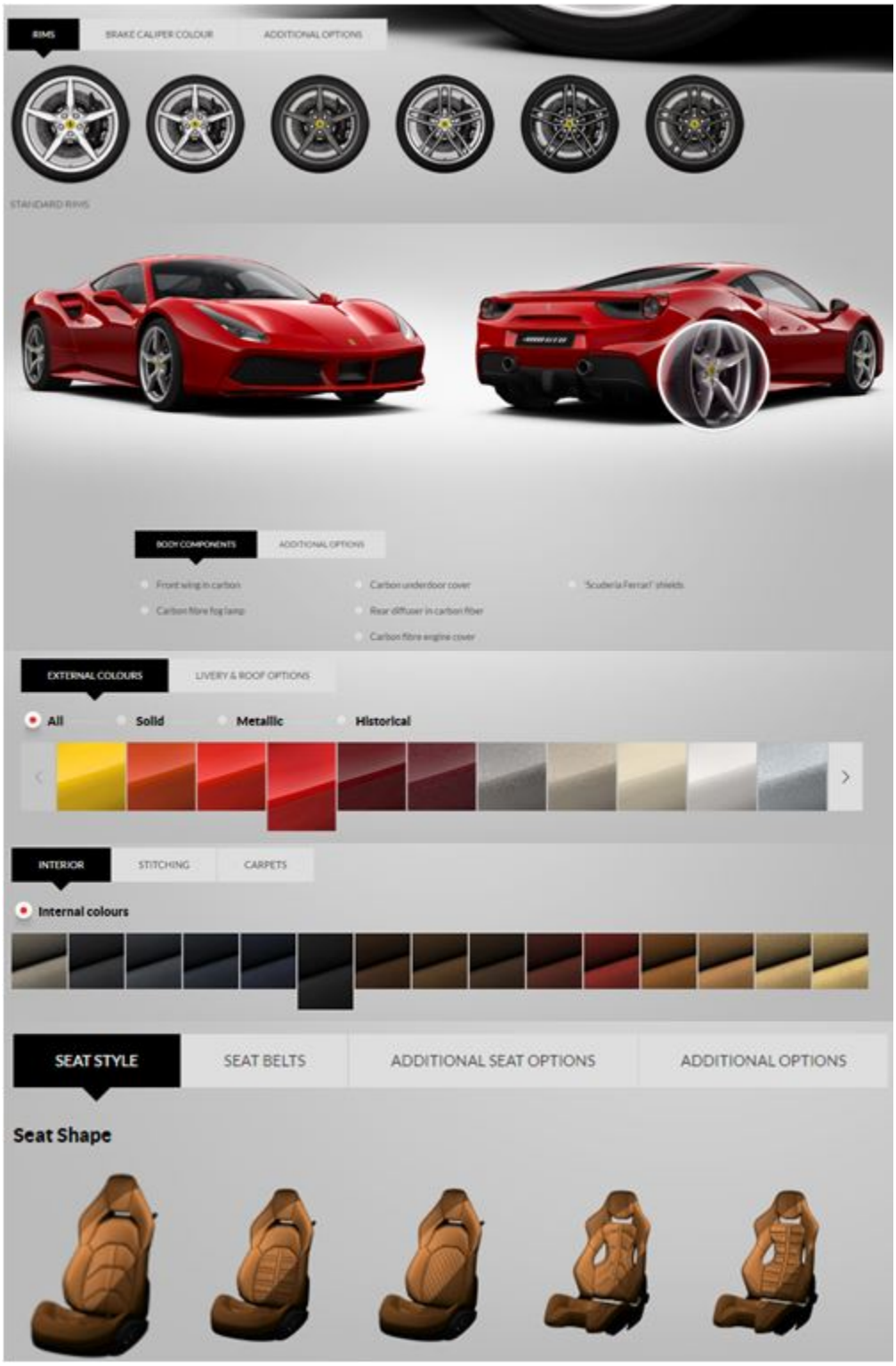

Figure 1: Example of available options during the mass customization process. Images are taken from `htpp://car-configurator.ferrari.com`

the number of variables and the number of clauses. Following this, each line represents a particular clause. A clause consists of the index of positive literal and the negative index of negative literal. The indices of variables are strictly positive integers (may or may not start from 1) while value 0 is used at the end of each line to terminate the definition of a clause. The following is the DIMACS notation for formula F with given in Section 2 with 3 variables and 2 clauses:

```
c simple.cnf
c
p cnf 3 2
2 -1 0
3 -2 1 0
```

3 Your Task

Given the ECC customizations as CNF Boolean formulas, your goal is to determine whether there exists an assignment of values satisfying the formula. If such as assignment exists, the customization is feasible (SAT), and otherwise, it is infeasible (UNSAT). A naïve idea to solve this problem is to exhaustively enumerate all possible variable-value assignments. Another option is to generate-and-check random assignments and hope for the best. Obviously, these ideas are not practical.

Unlike the simple example above, ECC has to tackle a large set of options. The instances consist of thousands of variables and clauses. As discussed in the lectures, an effective way of detecting feasibility is to use the DPLL algorithm. Your task is to implement a satisfiability solver based on the DPLL algorithm. The general outline of the DPLL algorithm leaves room for crafting efficient data structures as well as designing effective search heuristics. You are encouraged to experiment with different strategies and share your findings in the project report. Notice that, some instances might be feasible while some others are infeasible. Therefore, your solution should be able recognize infeasible instances and report them as such, and for feasible instances, it should return a satisfying value assignment.

You can assume that a *reasonable* time limit will be used to test the instances. Reasonable time is defined as the time it takes to leave your desk and get a fresh cup of coffee, which is roughly 5 minutes of running time for each instance on the department machines.

4 Support Code

We provide you with a data parser that reads CNF files (DimacsParser.java) and returns a simple SAT instance object (SATInstance.java). This way, you can immediately start attacking the heart of the problem. The support code is written in Java. The simple SAT instance class stores the input file as a set of integers, denoting varibles (`Set<Integer> vars`), and a list of set of integers, denoting clauses (`List<Set<Integer>> clauses`).

You are free to use the support code as is, tweak the data structures as you see fit, or discard it completely. If your choice of programming language is different, the support code can serve you as a reference.

Assuming you are in the `project` directory,

To compile the support code, type:

```
> ./compile.sh
```

To run the support code in a given instance, type:

```
> ./run.sh input/simple.cnf
```

In our running example, the expected output of the support code is:

```
> ./run.sh input/simple.cnf
Number of variables: 3
Number of clauses: 2
Variables: [1, 2, 3]
Clause 0: [1, -3]
Clause 1: [-1, 2, 3]
Instance: simple.cnf Time: 0.01
```

5 Output

Given an input file as its first command line argument via the `run` script, your solution should print out the result on its **last line**. You can print other statements before the result but they will be discarded. We suggest you to keep such debug information to a minimum since writing I/O is expensive. We use `stdout` for output and ignore other streams. Your submission will be tested on department's linux machines and it should work without any other software package installation.

In case of a feasible instance, the expected output is "Instance:" prefix followed by the instance name, "Time:" prefix followed by the number of seconds it takes your program to solve this instance given in 2-digit precision, "Result:" prefix followed by "SAT", and "Solution:" prefix followed by the satisfying value assignment for each variable.

```
./run.sh input/simple.cnf
..running..bla..
..bla..
Instance: simple.cnf Time: 1.23 Result: SAT Solution: 1 true 2 true 3 false
```

In case of an infeasible instance, use the "Result:" prefix followed by "UNSAT" and skip the solution tag.

```
./run.sh input/infeasible.cnf
..running..bla..
..bla..
Instance: infeasible.cnf Time: 1.23 Result: UNSAT
```

Finally, to run a benchmark on all instances in a given input folder using a fixed time limit while collecting the result (i.e. the last line) to a log file, type:

```
> ./runAll.sh input/ 300 results.log
```

Please make sure to follow this convention *exactly*! Failing to do so will cause evaluating your submission incorrectly.

6 Submission

```
/solution
|_ src/
|_ compile.sh
|_ run.sh
|_ runAll.sh
|_ results.log
|_ report.pdf
|_ team.txt
```

- Submit all your source code under the (`/src`) folder. Remember that commenting your code is a good practice so that one can follow the flow of your program at a high-level.
- Submit your (possible updated) `compile` and `run` scripts which compiles your solution and runs it on a given instance respectively. Make sure that the *last line* of your solution output follows the specification above.
- Submit the last line found for all the instance in the input directory in the `results.log`.
- Submit a short report in pdf format (1 or 2 pages max) that contains a brief discussion of your solution strategy, what worked and what did not, implementation techniques, and experimental observations. Please also include a note on how much time you spend on this project. This information is completely irrelevant to evaluation and kept solely for statistical purposes.
- Submit the cs login(s) of the team in `team.txt` listing one member per line in lowercase.

Have questions? and/or need clarifications? Please do not hesitate to contact us, your instructor and TAs are here to help. Good luck on your quest for helping ECC!