Mini Project – Backbones

This assignment requires you to implement an algorithm for finding the *backbone* of a propositional formula. The backbone ($\mathcal{B}_F$) of a formula F is the set of all literals that are true in every satisfying assignment of F.

1 Theoretical Definitions

Definition 1 (Backbone Literal)**.** *Let F be a satisfiable formula. A literal ℓ is a backbone literal of F if and only if $\mathcal{I}(\ell) = \top$ for every model $\mathcal{I}$ of F.*

Definition 2 (Backbone)**.** *For a satisfiable formula F, its backbone is the set of all its backbone literals.*

For example, for the formula $F = (\neg a \vee \neg b) \wedge (a) \wedge (c \vee \neg d)$, the literals a and $\neg b$ form the backbone of F. This means that every model of F must assign true to a and false to b.

The following proposition shows a simple way to decide whether a literal is a backbone literal:

Proposition 1. *Let F be a satisfiable formula and $x \in var(F)$. Consider the modified formulas $F_P = F \wedge (x)$ and $F_N = F \wedge (\neg x)$. Then one of the following holds:*

1. *F_P is satisfiable and F_N is unsatisfiable. Hence the literal x is a backbone literal of F.*
2. *F_N is satisfiable and F_P is unsatisfiable. Hence the literal $\neg x$ is a backbone literal of F.*
3. *Both F_P and F_N are satisfiable. Hence neither x nor $\neg x$ is a backbone literal of F.*

2 The Task

Your primary task is to implement the following iterative algorithm (see Algorithm 1) in `pySAT` to identify the backbone of a satisfiable propositional formula.

2.1 Implementation Details and Constraints

Your implementation must adhere strictly to the following requirements:

- Single Incremental Solver: Your code must use a single instance of an incremental SAT solver (e.g., `Cadical195` in `pySAT`) throughout the entire execution. You must not re-initialize the solver.

Algorithm 1 Iterative backbone algorithm

1: **Input**
2: F Satisfiable formula in DIMACS format
3: **Output**
4: $\mathcal{B}_F$ Backbone of formula F
5: $(answer, \nu) \leftarrow SAT(F)$ ▷ Find an initial model. ν is a satisfying assignment of F
6: $\Lambda \leftarrow \nu$ ▷ Initialize the set of backbone candidates with the literals from the initial solution ν
7: $\mathcal{B}_F \leftarrow \emptyset$ ▷ Initial set of backbone literals
8: **while** $\Lambda \neq \emptyset$ **do**
9: $\ell \leftarrow$ pick a literal from Λ ▷ Select and remove a literal ℓ to test
10: $(answer, \nu) \leftarrow SAT(F \cup \{\neg\ell\})$ ▷ Test if ℓ must be true in F by assuming $\neg\ell$
11: **if** $answer$ = UNSAT **then** ▷ Backbone found. ℓ must be true in all models
12: $\mathcal{B}_F \leftarrow \mathcal{B}_F \cup \{\ell\}$ ▷ Add ℓ to the backbone set
13: $F \leftarrow F \wedge (\ell)$ ▷ Add the unit clause (ℓ) to F to simplify future checks
14: **else**
15: $\Lambda \leftarrow \Lambda \cap \nu$ ▷ Filter the set of backbone candidates by the new model
16: **end if**
17: **end while**

- Solving under assumptions (Line 10): The satisfiability tests of the form SAT($F \cup \{\neg\ell\}$) must be performed using the `assumptions` argument of the `solve()` method in `pySAT`.
- Formula refinement (Line 13): When a literal ℓ is identified as a backbone literal, it can be permanently added to the formula F as a new unit clause in order to simplify all subsequent SAT checks. Use the solver's `add_clause()` method here.
- Filtering (Line 15): When the query is satisfiable, the new model ν returned by the solver is used to filter the set of backbone candidates Λ, i.e., Λ is reduced to only include literals that are also true in the newly found model ν. (If Λ and ν are sets of literals, this is a standard set intersection).

Note that, based on Proposition 1, whenever SAT($F \cup \{\neg\ell\}$) is UNSAT, it corresponds to Case 1 (i.e., F_P is satisfiable, while F_N is unsatisfiable), confirming that ℓ is a backbone literal. When SAT($F \cup \{\neg\ell\}$) is satisfiable, it corresponds to Case 3 (i.e., both F_P and F_N are satisfiable), which implies that neither ℓ nor $\neg\ell$ is a backbone literal. In this case, the discovered model also serves as a counterexample for all candidates in Λ that it falsifies, leading to the necessary filtering step.

2.2 Input and Output format

Your program will take as input a single argument: the path to a satisfiable formula in the DIMACS format. Your output must consist of two lines:

1. Backbone line: Starting with "`b`", list all the found backbone literals as a sequence of decimal numbers separated by spaces, followed by "`0`".
2. Count line: Another line starting with "`c`", followed by a single space and then the number of backbone literals that your program found.

The following example illustrates the expected input and output format:

```
c mini_example.cnf
p cnf 4 3
-1 -2 0
1 0
3 -4 0
```

```
b -2 1 0
c 2
```

Please follow these instructions regarding the output format carefully, otherwise your solution will be considered incorrect by the auto-checker. Note that the order of the backbone literals is not relevant, but in the example solutions they are sorted in increasing order.

3 Test instances

You can find a set of problem instances in TUWEL. The problems (and their example solutions) are in three folders:

1. Folder `test_instances/`: Test instances in DIMACS format.
2. Folder `test_solutions/`: The output for each of the test instances, adhering to the expected output format. Use these to check the correctness of your implementation.
3. Folder `eval_instances/`: Additional problem instances that you need to solve and submit your found solution (see details below).

The provided instances are rather small; they should all be solvable in less than 3 minutes (depending on your hardware).

4 Solution Submission

You need to submit your implementation and the output produced by your code for the problem instances in the `eval_instances` folder.

1. Please implement your solution as a single Python script file adhering to the following file naming convention:

   `backbone_<surname>_<matrnr>.py`

   where `<surname>` is your surname, and `<matrnr>` is your matriculation number.

2. For each problem in the `eval_instances` folder, create exactly one text file that contains the output of executing your program on it. The base name of the output file should be the same as the problem, but the extension of it should be "`.out`" instead of "`.cnf`".

All the files must be placed into a **single compressed file** (zip, tar or tar.xz) and uploaded to TUWEL before the submission deadline.