ECE650 Programming Assignment 2: Tseitin's Transformation

The skeleton for this assignment is available at the master branch of `https://git.uwaterloo.ca/ECE650-F2022/skeleton` in directory `pa2`. Follow the instructions in Assignment 0 to correctly fetch and merge the files from the skeleton!

This is the second in a series of assignments that is part of a single large project. The project is to build a SAT solver, and the purpose of this assignment is transform a Boolean formula F into an equisatisfiable formula F' in CNF using Tseitin's Transformation.

Your Program

For this assignment, you need to write a program that

- Transform a Boolean formula F (first use your PA1 formula parser to parse a formula string input) into an equisatisfiable formula F' in CNF using Tseitin's Transformation. The program is expected to output the satisfiability result of F by use of a SAT solver on F'.
- Continuously take input from standard input, and output to standard output. Errors shall also be output to standard output, but should always start with "`Error:`" followed by a brief description. Your program should terminate gracefully (and quietly) once it sees `EOF`. Your program should not generate any extraneous output; for example, do not print out prompt strings such as "`please enter input`" and things like that.
- Write your code in C++.
- Ensure that it compiles with the C++ compilor on

  `eceubuntu.uwaterloo.ca` and runs on `eceubuntu.uwaterloo.ca`. (Use `eceterm.uwaterloo.ca` to log-in from off-campus; then follow the instructions to connect to `eceubuntu`.)
- You can modify the skeleton as you wish, but you may only `#include` the following libraries (and no others!) for the current assignment: `iostream`, `fstream`, `sstream`, `iomanip`, `memory`, `iterator`, `string`, `utility`, `exception`, `regex`, `algorithm`, `cctype`, `cstring`, `vector`, `stack`, `list`, `deque`, and `map`.

Context-free Grammar for Valid Inputs

One valid input formula shall be a string generated from the Context-Free Grammar below:

```
Formula ::= ConjTerm | ConjTerm '+' Formula
ConjTerm ::= Term | Term '*' ConjTerm
Term ::= VarName | '-'Term | '(' Formula ')'
VarName ::= a continuous sequence of letter (upper- or lowercase) or digits (0-9);
the first character cannot be a digit; the total length shall be no longer than 10
```

* Blue represents non-terminals and red represents terminals.
* Arbitrary amounts of whitespace are permitted before, after, or in between any of these terms.

Semantics of the Input

One valid input is a string representation of a Boolean formula S_F in conformity with the aforementioned CFG.

In S_F, a Boolean variable is a sequence of letters (either in upper- or lowercase) or digits (0-9); however, the variable name cannot start with a digit and its total length cannot be longer than 10. "$+$" represents the 2-argument infix Boolean function OR; "$*$" represents the 2-argument infix Boolean function AND; "$-$" represents the prefix 1-argument Boolean function NOT. The order of operations is: $parenthesis > NOT > AND > OR$.

Tseitin's Transformation

Similar to Boolean evaluator, Tseitin's Transformation works recursively on the tree representation of a Boolean formula by traversing tree nodes. See in the course slide "normal-forms-solver" for how to conduct Tseitin's Transformation. The skeleton code stores the resulting CNF in the format of `std::vector<std::vector<int>>`: each inner vector represents a clause and each element in the inner vector is a literal; the outer vector represents the conjunction of clauses. This way of storing of CNF formulas is only a suggestion, not a requirement. We do not test on the intermediate results of CNF formulas; we only check the final satisfiability results.

Transformation Example 1

Input S_F:

$$(v1)$$

CNF F':

$$[1]$$

Comments:
1 represents variable $v1$ in this example.

Transformation Example 2

Input S_F:

$$c + d$$

CNF F':

$$[-1, 2, 3][-2, 1][-3, 1][1]$$

Comments:
2 represents variable c and 3 represents variable d in this example.

Transformation Example 3

Input S_F:

$$-b$$

CNF F':

$$[-1, -2][1, 2][1]$$

Comments:
2 represents variable b in this example.

Transformation Example 4

Input S_F:

$$c * (d + c)$$

CNF F':

$$[-3, 4, 2][-4, 3][-2, 3][-1, 2][-1, 3][-2, -3, 1][1]$$

Comments:
4 represents variable d and 2 represents variable c in this example.

* See provided executable `sample-helper-tseitin` for more examples.

SAT Solver

You need to use a SAT solver to check the satisfiablity of the transformed formula F'.

We will be using MiniSat SAT solver available at `https://github.com/agurfinkel/minisat`

MiniSat provides a `CMake` build system. The build process creates an executable `minisat` and a library `libminisat.a`. You will need to link against the library in your assignment (you can refer to the provided `CMakeLists.txt`).

Sample files are available in `https://git.uwaterloo.ca/ece650-f2022/skeleton/-/tree/master/sample-code/minisat-example`

Sample Run

The executable shall be called `pa2`.

```
$ ./pa2
a123
sat
a+c
sat
a* -- -a
unsat
(( -a)+(a*b)) *
Error: invalid input
 (1+2+3)*(1+2+ -3)*(1+ -2 +3)*-1
Error: invalid input
```

You can see test cases in `test\test0.in` and expected outputs in `test\test0.out` for more examples. Also, you can play with the provided sample executable `sample-pa2` on `eceubuntu`.

Unit Test

`doctest` is a single-header testing framework. We have provided the `doctest` header file, a sample `test.cpp` file of unit testing using `doctest`, and guidance on compiling the test executable in `CMakeLists.txt`. This unit test framework is just for your assistance and not a requirement.

Error Handling

You program should be able to identify invalid inputs according to the CFG and print an error message starting with "`Error:`".

Marking

Your output has to perfectly match what is expected. You should also follow the submission instructions carefully. It discusses how to name your files, how to submit, etc. The reason is that our marking is automated.

- Does not compile/make/crashes: automatic 0
- Your program runs, awaits input and does not crash on input: + 20
- Passes Test Case 1: + 20
- Passes Test Case 2: + 20
- Passes Test Case 3: + 15
- Passes Test Case 3: + 10
- Correctly detects errors: + 10
- Programming style: + 5

CMake

As discussed below under "Submission Instructions", you should use a `CMakeLists.txt` file to build your project. We will build your project using the following sequence:

```
cd pa2 && mkdir build && cd build && cmake ../ && make
```

If your code is not compiled from scratch (i.e., from the C++ sources), you get an automatic 0.

Submission Instructions

You should place all your files at the `pa2` directory in your GitLab repository. The directory should contain:

- All your C++ source-code files.
- A `CMakeLists.txt`, that builds a final executable named `pa2`
- A file `user.yml` that includes your name, WatIAM, and student number.

See `README.md` for any additional information.

The submitted files should be in `pa2` directory in the `master` branch of your repository.

You should assume that MiniSat will be placed in directory `pa2/minisat`. Your submission should include **only** your own code (including code provided by us in the skeleton). If your code is not compiled from scratch (i.e., from the C++ sources), you get an automatic 0.

Bonus

You can earn an extra credit of up to 10 points by implementing your own SAT solver instead of calling MiniSat solver. The expected behavior would be the same with that of PA2. You still need to finish PA2, i.e., PA2 will be graded using a universal criteria, regardless whether you plan to attempt the extra bonus. No skeleton will be provided for the bonus component.

To earn the extra points, you need to:

- Make sure an executable named `bonus` shall be built from `CMakeLists.txt` in a folder named "bonus" in your GitLab repo.
- You cannot use a truth table method to fulfill the task of SAT solving
- You need to submit a pdf file named `bonus-description.pdf` describing the algorithm you implemented for SAT solving
- The bonus mark will be graded based on the sophistication of your algorithm and the efficiency of your solver performance.
- No copying from existing solver codes is allowed; plagiarism in the bonus part will also affect your grade in PA2.