Write-up

Daniel McCormick and Aditya Arora

Tseitin Transform

The biggest challenge in this problem was handling the parsing of the clause. The troubles in this problem arose from mismatched brackets and other operator errors.

`Tseitin`:

1. Pre-process & detect errors in file [Mis-matched brackets and multiple operators]
2. Replace every number in string with corresponding symbolic variable
3. Generate Sub-formulas using depth
4. Introduce equivalence conditions and new variables
5. Simplify equivalences to CNF
6. Generate final CNF file

A 3rd party, symbolic library was used to make the process of dealing with parse trees easier in python, and not have to do it from scratch. Once the clause was parsed and Tseitin had been run over it, the only real challenge was reducing the parse tree to its CNF form.

Our output was verified by using the libraries "equivalent" function to ensure that no error was made in our transformation process.

As was required in the spec, the space complexity of Tseitin is linear for generating the equivalence clause. The run-time for the Tseitin part of the program is linear in the number of sub-clauses, but the simplification of the clause to CNF is exponential $O(2^n)$ because of the recursive way the (sub)trees are resolved level by level using the top down approach, going down to root each time.

DPLL

Our output was verified by running it on CNF files found online, as well as Prof. Ward's CNF files from ECE108. Special shoutout to the weird makefile, but `make` should build it, with some sort of output. Largely speaking, the most important thing is that fatures were built to spec.

The parser will try to throw errors if it can find simple formatting errors instead of recovering. Most online sat solvers discovered as roughly as picky - they don't want to speculate on the intent of the user. It will be generous if it can with error messages. The only additional functionality supported was it can take arbitrarily large lists without validing the header line. This was an intentional design decision to allow for convenient test case generation (in case someone modified the test case by hand), as discussed in our presentation. In particular, it can return the following errors:

1. Bad File (can't open for some reason)

2. Missing Header (required for DIMACs format, but can be dodged simply by starting with a 'p')

3. Invalid Line Format (ie invalid character, doesn't end with a '0')

Some critical assumptions made include an empty set is immedietely unsat, and an empty formula is also unsat. `dpllOpt()` is deprecated, and `dpll()` is what's used. The formula will also return without a full assignment if the current assignments can satisfy the formula. The formula is represented as a vector of vector (of integers). This does conform with the format.

It passes variables by value recursively since formulas are modified on the fly. Poor allocator, but unfortunately that's not required in the spec, so we abused it to the best of our abilities.

BCP is implemented to run in linear time, and does physically mutate the formula (removing all satisfied clauses, and unsatisfiable variables). It can also return if the current assignment is satisfactory (pre-BCP), or conflicting, returning BCP_SAT, or BCP_UNSAT. BCP_UNSAT only means at a local scale of assignments it's unsat (and does not comment on formulas).

Satisfying assignment is also run in linear time, and is run every instance of the call.

The actual code flow is:

`DPLL`:

1. parse clauses
2. if error return false/throw
3. sort formula by clause size (in non-decreasing order)
4. else do PLP and BCP
5. if error, return false;
6. return DPLL_Inner

`DPLL_Inner`:

1. if conflict, return false
2. do BCP

3. if conflict, return false
4. if satisfying assignment, return true
5. if all assigned, return true. This shouldn't actually happen but is retained for historic reasons
6. Choose some unassigned variable X. X = true.
7. If DPLL_Inner, return true
8. Else, X = false
9. If DPLL_Inner, return true
10. else, return false.

While in theory, the worst-case performance for DPLL is O(2^n) and worst-case space performance is O(n), the facile implementation is very reckless with runtimes and allocation. The worst-case performance for this method is O(n2^n) and worst-case space performance is O(n^2). This is because each of the O(n^2) searches runs in O(n) time and can add O(n) memory, resulting in all runtimes to be O(n) times the idealized form.

Ways to improve this algorithm would be to pass-by-reference, and keep changes local, resulting in O(n) space. It'd also be better to track each clause, to tell if there's a conflict in constant time. Though this will likely have a very large initial overhead, the asymptotics will be a lot more favourable. Overall however, this is correct and conforms to the specification applied. As such, we are happy with the correctness of our code.

Klee (Bugs in `coreutils`)

The first few tutorials were really interesting, and taught me some really valuable lessons about the ways `klee` could be used.

- In tutorial 3, it was *really* fascinating to see `klee` generate the test cases that would break through pseudo-walls in the "maze"
- Moreover, Tutorial 4 where a password is cracked by running `klee` on the "checker" really made me aware of that fact that just complexifying code for humans, does not make it any difficult for computers to see right through the facade

The biggest challenge in this problem was actually the compilation of `klee`. The first few parts of the tutorial worked fine with the DOCKER container version of `klee`, **but** the actual `coreutils` parts of the assignment required compilation of `klee` from scratch on a virtual machine.

The number of hours spent on actually getting `klee` to compile are not measurable on one hand. A parallely running web-server had to be taken down because the compilation was capping out the RAM on the relatively cheap VM. Optimization had to be done for storage on the VM because all the installation repositories along with build files, capped out the storage as well.

`Klee`:

1. Clone `klee` and Build with `gcc`, `uiobc` and `POSIX`
2. Clone `coreutils` and build with `gcov` and `llvm`
3. Clone and install `wllvm`

Once the compilation was done, the actual "testing" of `coreutils` was actually extremely easy.

- `md5sum`: 1 ptr error
- `mkdir`: 1 model error, 1 ptr error
- `paste`: 10 ptr errors

Encode Graph Vertex Cover to CNF

This was the only problem that required some of our own thinking. After some discussion, the ripple adder solution was tried, but sadly we lacked enough context to be able to use the bits of our output and feed it into our comparator. An earlier version with this failed attempt is linked here

In the end, we resorted to using an exponential ($n!$) growth algorithm by generating all the possible combinations of sum and simplifying to get our sum. All the requirements from the original problem are satisfied.

The simplified result did reveal some insights on how we could optimize our algorithm to reduce time complexity, but the approach was not implemented due to lack of time.

The run-time of this algorithm has exponential growth $O(n!)$ in the number of nodes in the original graph since the encoding requires the generation of all combinations of products from size 1 to k, inclusive. This can be improved by possibly encoding an exclusion for the same based on the relation between k and n. The space complexity is also exponential $O(n! + a)$, where a is the number of edges in the input.

`EncodeGraphCover`:

1. All edges from the adjacency matrix are added to the CNF clauses
2. Generate all combinations of the sum for all values 1 to k
3. OR all the clauses generated above, and minimize
4. Convert overall clause to CNF
5. Write CNF file

Our final results for this algorithm were compared with the output of a more traditional vertex cover algorithm.
