SAT Solver Project Proposal

CSC 520 Spring 2022 001

Kasimir Schulz, Kelly Wang, Nathan Vercaemert

2022-03-22

Contents

1 Project Description

1.1 What is the problem?

Boolean satisfiability (SAT) represents the idea of matching a sentence of propositional logic in conjunctive normal form (a Boolean formula) with a corresponding set of assignments to the propositional symbols that satisfies the sentence.

1.2 Why does it matter?

- General: The SAT problem was the first problem proven to be NP-Complete (Cook-Levin theorem). Every problem in NP can be represented as a SAT problem. Many problems in computer science are in NP, and a SAT solver allows an alternative solution (sometimes the only known solution) to these problems for comparison or ease of implementation. By implementing our own SAT solver, we can more comfortably approach the issue of converting a problem into a form for SAT solving. By understating the input format and implementation limitations of our SAT solver, we can more easily utilize SAT-solving functionality.

- Optimizations: For example, an addition problem with just 4 bit integers requires hundreds of connections (with tens of thousands of connection checks) with our current implementation, and the number of connections scales exponentially.

1.3 What is our plan for addressing this problem?

Our textbook provides a comprehensive summary of the techniques used in state-of-the-art SAT solvers in section 7.6. We propose to implement these techniques in order of complexity in order to meet the intended scope of the project. An example of a simple technique that we will implement: outlined within the DPLL algorithm, "early termination" evaluates whether any clause within the CNF sentence is false at each incremental assignment in the model (backtracking if a clause is found to be false). As an example of a technique that is likely outside the scope of our project: the textbook mentions "clever indexing" referring to the idea of indices for clauses, based on various criteria and dynamically updated, that constitutes the most complicated technique for SAT solver optimization. We will consider these techniques to the extent that time allows. The techniques we will implement are further explored in the following section: Optimization Techniques in SAT Solver Context.

1.4 Why do we think this is the appropriate way to address this problem.

In an reasonable appeal to authority, we are assuming that our textbook considers the most important and consequential techniques of SAT solver optimization.

2 Optimization Techniques in SAT Solver Context

2.1 Early Termination

See What is our plan for addressing this problem?.

2.2 Pure Symbol Heuristic

In the CNF representation of the Boolean formula to be solved, if a literal only ever appears with the same sign, it can be assumed to have an assignment that makes the literal true.

2.3 Unit Clause Heuristic

If a model has been assigned such that a clause only has one unassigned literal remaining, and the clause's truth relies on that literal (everything else is false), then we can assign the value to that literal that makes the clause true.

2.4 Component Analysis

If the clauses of the CNF formula can be divided into disjoint subsets that share no propositional symbols, we can solve each subset individually more efficiently.

2.5 Variable and Value Ordering

The "degree heuristic" (in CSP context) can be used to determine the order in which propositional symbols are assigned values. By choosing the propositional symbol that appears most frequently, we can more efficiently solve a SAT problem.

2.6 Intelligent Backtracking

By keeping track of combinations of assignments that prove problematic, we can ensure that these combinations are not re-assigned after backtracking. There is a balance between the number of problematic assignments kept and the efficiency of the algorithm. We plan to explore this by allowing this value to be adjusted. Conflict-directed backjumping implements a form of intelligent backtracking by keeping track of sets of assignments that do not allow progress. This will likely reach the scope of our project as Random Restarts and Clever Indexing introduce significant levels of complexity.

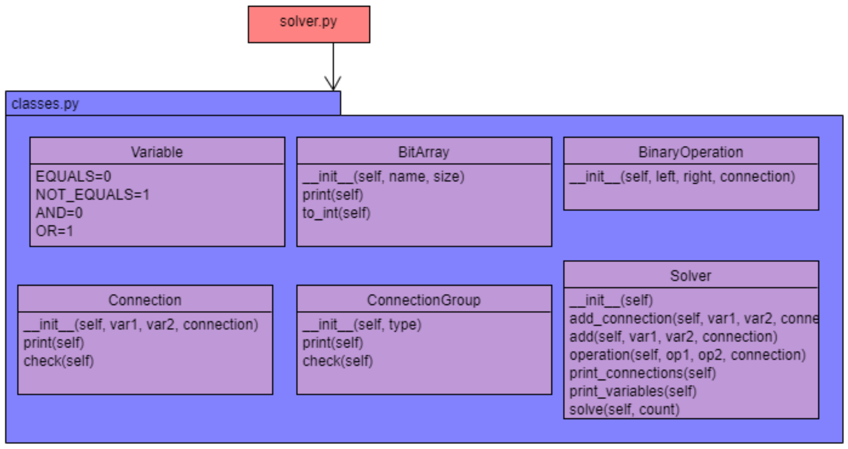
2.7 Random Restarts

Metrics can be used to track whether or not the solver is making progress towards a solution. If it is determined that the solver is no longer making significant progress, solving can be restarted with a known (dynamic) set of guaranteed assignments and a random selection of which propositional symbol to assign next.

2.8 Clever Indexing

This set of techniques for SAT solver optimizations is broad and outside the scope of our project. We will venture into these optimizations time-permitting. An example of clever indexing: by keeping track of the clauses in which particular propositional symbols appear with only their positive literals (updated dynamically based on which clauses have already been satisfied by previous assignments), we can further the "degree heuristic" to include the notion of a probabilistic weight that an assignment might falsify the model.

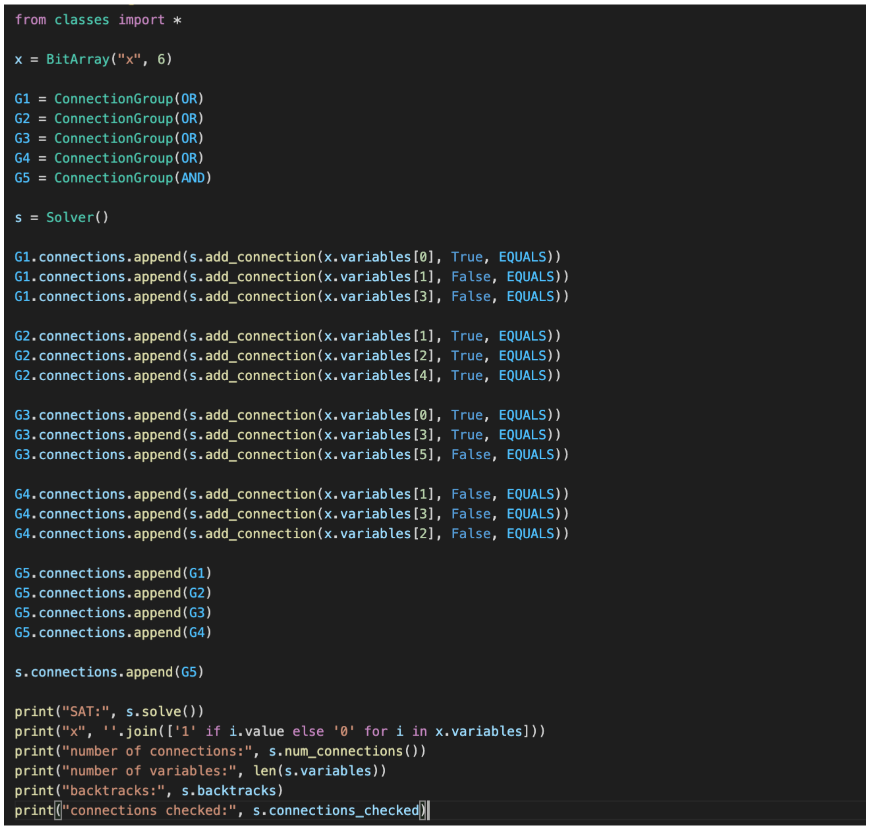
3 Sample Design



4 Sample Main Class

Here we have an example of our current implementation solving a 3-SAT problem.

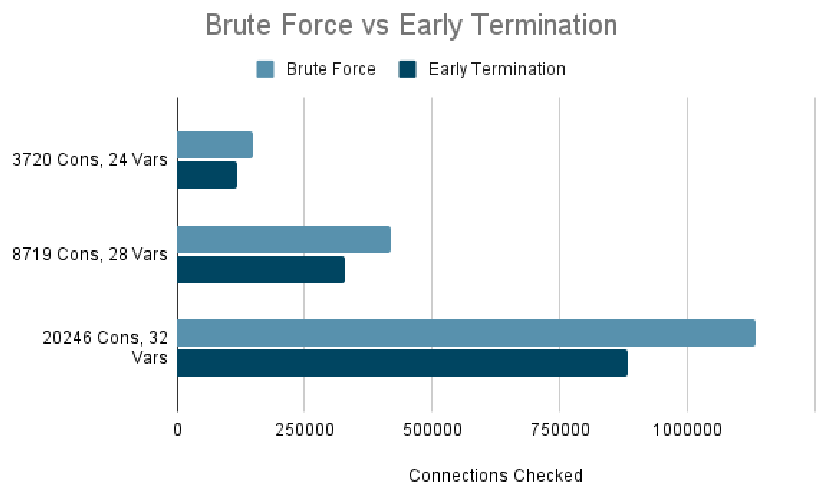
$(x1 \lor \neg x2 \lor \neg x4) \land (x2 \lor x3 \lor x5) \land (x1 \lor x4 \lor \neg x6) \land (\neg x2 \lor \neg x4 \lor \neg x3)$



```python
from classes import *

x = BitArray("x", 6)

G1 = ConnectionGroup(OR)
G2 = ConnectionGroup(OR)
G3 = ConnectionGroup(OR)
G4 = ConnectionGroup(OR)
G5 = ConnectionGroup(AND)

s = Solver()

G1.connections.append(s.add_connection(x.variables[0], True, EQUALS))
G1.connections.append(s.add_connection(x.variables[1], False, EQUALS))
G1.connections.append(s.add_connection(x.variables[3], False, EQUALS))

G2.connections.append(s.add_connection(x.variables[1], True, EQUALS))
G2.connections.append(s.add_connection(x.variables[2], True, EQUALS))
G2.connections.append(s.add_connection(x.variables[4], True, EQUALS))

G3.connections.append(s.add_connection(x.variables[0], True, EQUALS))
G3.connections.append(s.add_connection(x.variables[3], True, EQUALS))
G3.connections.append(s.add_connection(x.variables[5], False, EQUALS))

G4.connections.append(s.add_connection(x.variables[1], False, EQUALS))
G4.connections.append(s.add_connection(x.variables[3], False, EQUALS))
G4.connections.append(s.add_connection(x.variables[2], False, EQUALS))

G5.connections.append(G1)
G5.connections.append(G2)
G5.connections.append(G3)
G5.connections.append(G4)

s.connections.append(G5)

print("SAT:", s.solve())
print("x", ''.join(['1' if i.value else '0' for i in x.variables]))
print("number of connections:", s.num_connections())
print("number of variables:", len(s.variables))
print("backtracks:", s.backtracks)
print("connections checked:", s.connections_checked)
```

5 Example Efficiency Increase

In our naive implementation we were able to easily implement the "Early Termination" employed by DPLL. This graph shows the efficiency gains that were made with this implementation. For the purposes of this graph, a connection is the same thing as a constraint.


Brute Force vs Early Termination
Brute Force
Early Termination
3720 Cons, 24 Vars
8719 Cons, 28 Vars
20246 Cons, 32 Vars
Connections Checked