SAT SOLVER- END SEM PROJECT

Mukul Verma(160101044)
CS 508 - Optimization Methods

May 30, 2020

1 Idea

In this report, we will design a SAT-Solver based on **conflict-driven clause learning (CDCL)** approach. The solver presented in this report is a *clause learning, non chronological backtracking based SAT Solver*. We will use CDCL instead of DPLL (**Davis-Putnam-Logemann-Loveland**) approach for our SAT solver as CDCL solvers treat each conflict an opportunity to learn more about the SAT instance. Each time a conflict is identified, the solver extracts the reason as a clause by some learning scheme and adds it into the clause database to avoid recurrence of that conflict and help prune the search space in the future. This process is called **conflict analysis** or **learning**, and the newly added clause is called a **learnt clause**. Guided by the learnt clause, the solver then backtracks to some earlier level which may not be the one with the most recent unflipped decision, potentially pruning a larger portion of the search space. This kind of backtracking is called **non-chronological backtracking**.

In the implementation, the SAT solver is representation by a class *Solver* and the clauses are represented using the class *Clause*. The solver instance S holds the clauses in the form of list $S.cnf$ of the clauses objects. The clause object C holds a clause $C.c$ which stores the list of literals it has. All the clauses which are not unit are pushed into that list whereas the unit clauses are pushed into another lists $S.unit\_c$. The reason behind this is that the literals which are present in the unit clause are always assigned $True$ and they do not have any watch literals as well. The another reason is also that is reduces the search space of the sat solver. The sat solver also maintains a trail $S.trail$ which contain the assignments of the literals in the sat formula. The unit clause literals are also not pushed into the trail as their value is never changed and hence need not to be tracked.

We note that learning does not change the fact that the worst case time complexity remains exponential in terms of the number of variables. However, in the case of some classes of real applications, a good implementation shows an acceptable time complexity when combined with appropriate heuristics. Based on this idea we will present the following procedures used in the proposed SAT Solver and the heuristics used in them to increase the performance of the proposed SAT Solver.

1.1 Boolean Constraint Propagation (BCP)

Experiments show that for most SAT instances, a major portion (70% 90%) of the solvers' runtime is spent in the process of Boolean constraint propagation (BCP). Therefore, implementing an efficient BCP is key to any SAT solver.

In our implementation, the literal which need to be propogated using BCP are maintained or stored in a stack (implemented using a list) called BCP_Stack. We use stack as the sat solver searches the space in the DFS manner. After performing the unit propogations, BCP returns the state of the Solver. If the returned state is $Solver.UNSAT$, that means that the cnf formula is UNSATISFIABLE and if the returned state is $Solver.CONFLICT$, it means conflict is encountered and conflict analysis procedure is called. We have used following heuristics to improve the performance of the BCP procedure

1.1.1 Two Watched Literal

Chaff (Engineering an Efficient SAT Solver) proposed proposed a lazy data structure, the watched literals, to improve the performance of the BCP. We have used this scheme because whenever a literal is unit propagated, we need not to check all the clauses in which that literal is present. We only check the clauses in which this literal is one of the watch literal. Assuming there are n clauses, this method works with additional $2n$ space, which is much smaller than the naive methods. Besides, it incurs no overhead in backtracking. Hence, this approach is efficient.

In the implementation, the left and the right watch literals for a clause C are represented by using $C.lw$ and $C.rw$ respectively. Each literal l maintains a list of the clauses in which this literal is a watch literal known as *watch list*. We uses an unordered dictionary to store the clauses of the watch literals. The key of the dictionary is the literal l and the value is a list of the indices of the clauses in which this literal is a watch literal. The watch list for a literal gets updated during BCP procedure.

1.2 Conflict Analysis

The conflict analysis procedure learns a clause called *learned clause* and add it to the database of the clauses which is $S.cnf$ and $S.unit\_c$ depending upon whether the clause is unit or not. We will use the knowledge learnt from conflicts is also used to guide the backtracking procedure. The learning scheme is based on the following heuristics

1.2.1 1-UIP based Learning Scheme using Implication Graph

The solver identifies the reason of a conflict with the help of a directed acyclic graph (DAG) called *implication graph* as proposed in Zhang in his paper "Efficient Conflict Driven Learning in a Boolean Satisfiability Solver". In the implication graph, we call a node as UIP at level dl iff any path from the decision vertex at dl to the conflicting vertices needs to go through that node. The intuition behind 1-UIP is to find a reason closest to the conflict.

In the implementation, the implication graph is represented by the antecedent. Each literal maintains a antecedent clause which is basically a clause, because of which that literal get its value when that clause has become a unit clause. A list *S.antecedent* maintains this information. The antecedent of decision literals is -1.

The implementation first selects the current clause as the conflicting clause and adds the literals not in the conflicting level (at which conflict occurs) to the learned clause and mark the literals of conflicting level in the current clause. Then, the literal which is last assigned literal is resoluted with its antecedent clause. This resoluted clause is then made as the current clause and same procedure is continued till we reaches to a literal which is the only marked literal. This literal becomes the 1-UIP in the implication graph and its negation is added in the learned clause. The second highest decision level of the literals in the learned clause is made as the back-jump level.

The reason behind using the 1-UIP scheme is that is learns a clause in which only 1 literal is of the conflicting level and hence after back-propagation, the learned clause becomes unit clause and we get a new literal to propagate just after the backtracking.

1.3 Backtracking

In our implementation, we uses non chronological backtracking. The backtracking level is determined by the learned clause which is the second highest level of the literal in the learned clause. The backtracking simply reset all the literal assignment just after the backtrack level. In the implementation, solver maintains the trail *S.trail* of the assignments of the literals . The data structure used for *S.trail* is a dictionary with keys as decision level and value as the list of literal assigned in that decision level in the order in which they are assigned. This trail helps the backtracking to reset the variables of those decision levels which needs to be backtracked.

1.4 Decide

Decision heuristic has a significant impact on the performance of the solver. Even for the same basic solver structure, different decision heuristics may produce search trees with drastically different sizes. Making a decision requires two steps. First, select a free variable. Second, assign the variable a selected value. In our solver we have used following two schemes one for each

1.4.1 Variable Selection Heuristic - VSIDS

In order to select a variable, we have used *Variable State Independent Decaying Sum (VSIDS)* proposed by Chaff in his paper "Engineering an Efficient SAT Solver". For each variable, VSIDS keeps a score and increases it once this variable involves in a conflict. Thus the score can be used to evaluate how active a variable is in the CNF formula: the higher score a variable has, the more active it is. In such a manner, the solver gives priority to assigning active variables.
2em In the implementation, the solver a dictionary *S.score_map* and a list *S.var_score*. *S.score_maps* has key as the score and the value as the set which contains the literals having that score. *S.var_score* maintains the score of each variable individually. This is needed when score of the variable is updated and can be done $O(logn)$ complexity where $logn$ comes from the insertion into the set. Apart from these two data structures a variable *S.max_score* and reverse iterator *S.score_map_r_it* is also maintained by the solver. *S.max_score* stores the current maximum score among all the unassigned variables. The advantage of maintaining the *S.max_score* is that there is not need to iterate from the last of the dictionary keys till the beginning, we can iterate from the *S.max_score* key as all the variables having score

greater than this are already assigned some value and hence cannot be used in decide. This reduces the searching cost. The reverse iterator *S.score_map_r_it* is used to iterate in the dictionary *S.score_map* in the reverse order, that is from bigger scores to smaller scores.

1.4.2 Variable Assignment Heuristic - Phase Saving

In our sat solver, we use *Phase Saving* approach to assign the values to the decided variable. Phase saving is a low-overhead caching technique that always assigns the free variable its last value. This approach is helpful as solutions of sub-problems can be reconstructed in a different order and the solver enjoys a decrease of work repetition.

In the implementation, we have used a list named *S.lit_prev_state* to store the previous assigned value of the variable. Whenever a variable is selected using VSIDS approach, the variable is assigned the value assigned in this list. If it is unassigned the variable is assigned FALSE value. This is done due to the fact that in the actual satisfiable assignment most of the variables are FALSE.

1.5 Restart

During the searching of the satisfiable assignments, the solver may get stuck in a complex part of the search space. Hence, we use Restart to eliminate this problem. We use static scheduling in order to decide when to do a restart. We uses following heuristic for the restart scheduling.

1.5.1 Static Scheduling based on Iterative two nested Luby series

In [23] Luby series is demonstrated to be very efficient. This series follows a slow but exponentially increasing law like $\{1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 4, 8, \ldots\}$. A factor called $S.LUBY\_FACTOR$ is used in the implementation to generate the the series of limits for the conflict interval. We have used $S.LUBY\_FACTOR = 100$. Hence the generated luby series for conflicts is $\{100, 100, 200, 100, 100, 200, 400, \ldots\}$.

For nested luby series, the solver maintains an inner limit *S.inner* and an outer limit *S.outer*. The outer series bounds the inner series. Whenever the number of conflicts since the last restart reaches $self.inner$, the solver performs a restart and *S.inner* is set to the next value of the inner series. Whenever *S.inner* reaches *S.outer*, *S.inner* is reset to the initial value of the inner series and *S.outer* is set to the next value of the outer series.

2 Data Structures

2.1 Watch List

The dictionary *S.watches* is used to store the watch list of each literal. This removes the requirements for converting the negative literals to some other encoding so that the literals can be mapped to the list indices. Hence this saves the time for function to convert literal into encoded form and also the encoded literal value back to original literal.

2.2 Trail

In the implementation we have used a dictionary *S.trail* to store the trail of the assignments. The key is kept as the decision level and value is the list of literals assigned in the order at the decision level. This approach is better than using a single list as it removes extra load of maintaining the indices where the decision variables are present and hence saves the space.

2.3 Scoring Map

In the implementation of scoring scheme of the variable in decide, we have used a sorted dictionary *S.score_map*. The keys are the score and values are the sorted map of the literals having that score. Note that maintaining the sorted dictionary, saves the time for searching the maximum score variable as maximum score always lies before some particular max score as all the variables after than are assigned. This fact is used to reduce the time complexity of searching by using a variable *S.max_score* which always points to the key in which maximum score variable can be found. The variables having greater scores are already assigned.
One more benefit of using map for storing variables is that it reduces the deletion cost. The deletion cost in map is $O(logn)$ where as in list it is $O(n)$. Though it introduces extra complexity of $O(logn)$ when inserting some variable as in case of list it is $O(1)$ which is not much penalty as $log(n)$ vs $O(n)$ for sufficiently large n.