Building A SAT solver

TEAM MEMBERS: ASHISH GONDIMALLA, VIGNESH RAGHAVAN

INTRODUCTION:

In this project we aimed to build a SAT solver (Option 1). We tried different ideas for data structures, methods to quickly compute results at each node of the search tree. We relaxed on memory usage and focused on compute time.

Techniques we used and implemented.

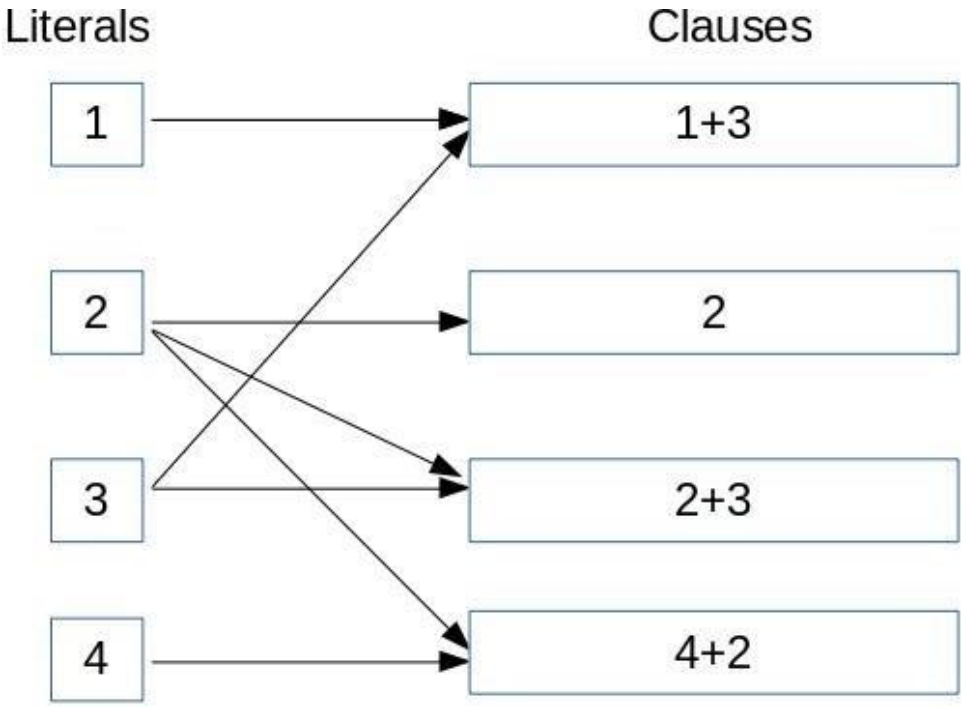
Lit Clause Database:

We choose a Lit clause database for faster literal assignment related operations.

Literals are stored as positive numbers. Negative literals in DIMACS are stored as odd positive integers and positive literals as even integers.

For every literal, we store a list of pointers to the clauses which contain that literal. When a literal l is assigned, the clauses pointed by that literal are set to satisfied. Among the clauses pointed by the complement of the literal ,we look for unresolved clauses which give forced decisions.

The structure is as follows:



Each clause in the database has counters – Scount (how many times it got satisfied) and UAcount (No: of unassigned variables left).

The use of counter for Scount simplifies the Undo technique. Scount is incremented when any of literal in the clause is assigned. Scount is decremented when the literal is undone. A non-zero Scount means a clause is satisfied. UNSAT clauses can simply be identified by looking for a zero Scount. While undoing, we can simply decrement Scount and state whether a SAT clause became satisfied due to the current undone literal assignment or was it satisfied even before.

BCP without watch literals:

When a literal is assigned, we get its complement and for all clauses pointed by complement having Scount of zero(I.e unresolved clauses), we decrement UAcount. If this UAcount reaches 1, we can identify a forced decision in the clause by looking for an unassigned literal. If it becomes 0, then we obtain a fully unsatisfied clause which implies a conflict. Thus we didn't need to use watch literals. Also, watch literals update will be time consuming. When we assign a watch literal, we have to scan the clause and get the next unassigned variable which is avoided here.

However, if a unit clause is found, the forced decision can be immediately given by watch pointer. At this point, we go to the clause database and try to deduce the forced decision. Also for checking for Satisfiability, we don't have to parse the clause. Instead, we can just look up for a nonzero Scount.

The savings in computation here was due to increased memory usage. We basically have two databases of same information , lit to clause and clause to lit databases (for faster BCP). We use appropriate one for a faster speed.

0-1 Lookup matrix for conflict clause:

For each conflict, our algorithm creates a conflict clause of only free decisions. For doing this quickly instead of a traversal every time, we maintain an implication graph with partially computed free decision clauses for every forced decision. When we hit a conflict, we do resolution (bitwise OR) of the columns of the literal l and its complement.

The procedure for this is to maintain a 0-1 literal-literal matrix. Whenever we make a free decision, we put a 1 in the corresponding complemented decision column. When we get a force decision or conflict, we take the clause and do column wise 'OR' operation for all literals in the clause (except the current forced or conflicting literal). This updates the current column with only free decisions responsible for this forced decision.

This operation is like memoization, where instead of computing all free decision literals, we use partially stored results.

VSID

VSID idea is borrowed from chaff. We use a conflict clause to update the vsid tag in each literal. For every 100 conflicts, we divide this count by 4. This is used to update vsid counts.

Then for branching, we look at all literals and choose the one with maximum vsid count. This helps in satisfying more conflict clauses.

Nonchronological Backtracking

While backtracking, we check if a free decision literal to backtrack is present in the conflict clause or not. If it is not present, we continue to undo till we reach such a decision.

Conflict learning and removal

We add every conflict clause to a separate list called lclauses. For removing clauses, whenever we hit the threshold value (500 for our case) we simply remove the first (oldest) 1/16th of the clauses from learnt clause queue.

Removing a conflict clause is computationally expensive for literal clause data structure (has complexity of O(n^2)). To reduce the cost it is better to put a separate list for learnt clauses in a literal. Also we try to keep learnt clause set to a small number (around 500) thus reducing the complexity of removing a clause.

Preprocesssor

We added a preprocessor to remove pure literals and unit clauses (obvious forced decisions.) For pure literals we check if any of the literals has 0 clauses attached and force the complement to 1. For unit clauses we look for UAcount and if it matches 1 then we force them.

The presolve() is a small solver with evaluate SAT and unresolved clauses. The only difference it operates on a queue Q. And evalSATclauses removes a given clause. Finally if a conflict occurs we return UNSAT.

PSUEDO CODE:

The core of the algorithm is as follows:

```
Stack S;
Queue Q;
backtrack = false;
S.push(Firstdecision());
while(!S.empty())
{
    if(!backtrack)
    {
        evaluateSATclauses(S.front())
        //evaluate unsat clauses also calculates forced decision and puts them //in queue
        conflict = evaluateUNSATclauses(S.front(),Q, ccl)
        if(!conflict)
        {
            If(checkSAT())return true;
            If(!Q.empty()) S.push(Q.pop());
            else S.push(decide());
        }
        else
        {
            Q.clear();
            learnconflict(ccl, decision);
            backtrack = true;
        }
    }
```

```
        else
        {
                //decisioninlearntconflict check whether a decision is present in conflict
                //clause and allows us to backtrack nonchronologically.
                if (!forced or !visited or decisioninconflictclause())
                {
                        backtrack = false;
                        S.pop();
                        Undo();
                        S.push(complement);
                }
                else
                {
                        Undo();
                        backtrack = true;
                }
        }
}
```

For evalUNSATclauses:

```
EvalUNSATclauses(lit l , Queue Q, clause * ccl)
{
        UpdateUAcountsofclauses();
        foreachclause
        {
                if(UAcount == 1)
                {
                        Q.push(Findforceddecision());
                }
```

```
            If(UAcount == 0)
            {
                ccl = currentclause;
                Updateimparray(ccl);
                return false;
            }
        }
        return true;
}
```

The implication 0-1 array is updated in findforceddecision, decide() too. Learnconflict() uses 0-1 array to compute the forced decision.

EvalSATclauses just updates Scount for clauses of current assigned literal.

RESULTS:

The SAT solver is evaluated based on the time it takes to solve the SAT/UNSAT instances for different benchmarks, with different features of the solver enabled/disabled using a switch. The solver types are classified based on these features as given below.

1. A solver with only BCP enabled and chronological backtracking (CBCP).
2. A solver with non-chronological backtracking but no conflict learning (NCBCP).
3. A solver with conflict learning and removal heuristics (NHCL).
4. A solver with VSID heuristic based free decision (HCL).

The above features of the Solver are evaluated for their improvements by studying their performance on 3 instances of each of the benchmarks – S100, S150 and U150, U175.

The following are the results (computation time in seconds) obtained for 4 different Solver types – CBCP, NCBCP, NHCL and HCL. For each solver type, the following statistics were obtained on 3 instances of SAT/UNSAT (as indicated by prefix S or U), for 2 different literal sizes.

Chronological Backtracking Only (BCP)

CBCP	1	2	3
S100	0.58	0.09	0
S150	1.54	82.7	39.86
U150	78.11	1066.39	1432.36
U175	4031.62	>9000	6596.46

Non-chronological backtracking (No conflict learning)

NCBCP	1	2	3
S100	0.46	0.06	0
S150	1.01	56.93	29.45
U150	58.99	819.9	593.92
U175	2043.48	8027.49	2958.76

Conflict learning and Removal

NHCL	1	2	3
S100	0.39	0.05	0
S150	0.75	45.9	20.91
U150	43.16	614.93	457.49
U175	1324.49	5533.88	1753.55

VSID heuristic based free decision (+ Conflict learning, removal)

HCL	1	2	3
S100	0.12	0.05	0
S150	0.74	7.91	11.77
U150	21.55	86.7	54.91
U175	129.89	545.45	332.84

The following result is obtained for different instances of each benchmark by varying the VSID saturation limit while the number of learnt clauses from a conflict capped to 500. And, 1/16th of which will be removed upon reaching this capped limit.

HCL with varying (VSID saturation limit)

HCL	1	2	3
U150 (20)	28.46	229.24	148.36
U150 (50)	18.27	114.96	74.14
U150 (100)	21.55	86.7	54.91
U150 (200)	15.72	63.2	35.46

The following result is obtained for different instances of each benchmark by varying the capacity of the allowed learnt clause. The VSID saturation limit is set to 100, upon which all VSIDs are divided by 4.

HCL with varying (Learnt clause capacity)

HCL	1	2	3
U150 (100)	13.71	73.52	42.8
U150 (200)	18.28	88.47	47.28
U150 (500)	21.55	86.7	54.91
U150 (1000)	16.21	91.73	53.87

Note:

We also tried Hanoi4.cnf which has **718** literals and **4934** clauses. It took about **15000** seconds. (~4hrs) with debug statements in NHCL. This is about the same time as hanoi4 for GRASP paper. Possibly without debug the solver NHCL might be faster than GRASP.

We finally tried with preprocessor and HCL without debug statements. The time for hanoi4 was now **2884.12 seconds (~45min)**.

ANALYSIS:

From the above results, we observe the following.

1. With the addition of each of the above techniques, the evaluation time was lesser.

   In terms of evaluation time, it was observed that for all the instances of all benchmarks (SAT and UNSAT), HCL < NHCL < NCBCP < CBCP.

2. Larger the literal size of the benchmark, longer it took to evaluate SAT/UNSAT.
3. UNSAT instances took longer to evaluate than SAT instances (trivial result).
4. Larger the VSID saturation limit, better is the performance of the HCL solver. We deduce that there should exist an optimum VSID saturation limit for which HCL solver will be fastest. Beyond that optimum limit, the performance of HCL solver saturates.
5. The capacity of the learnt clauses is a trade-off between the computation time in exploring the Search tree (for smaller capacity) against the computation time involved in managing the learnt clause database during the Search (for larger capacity). Hence, no general trend can be deduced from varying the learnt clause capacity.

CONCLUSION:

We realized that the search tree can covered faster with better heuristics. We have seen this trend for heuristics mentioned above. Among the heuristics the best values for parameters (like conflict clause queue length , vsid reset etc) can be determined statistically. Also, the 0-1 Lookup matrix might be computationally more intensive than the successive resolution of the conflict clauses.

We also planned to add features like preprocessing, pure literal elimination, Clause removal after backtrack at root, updated vsid (from berkmin). But we were unable to update them due to debugging issues. We could have seen improvements by adding them too.