Assignment 2 COMP6741: Parameterized and Exact Computation

Felix Abrahamsson - z5178509
David Coates - z3462785
Suganya Suresh - z3288066

September 20, 2017

Literature Review

1. *Describe the unit clause simplification rule, also known as unit propagation (UP). Give an example of a* SAT *instance where UP can be applied.*

   A unit clause is a clause containing only a single literal. If a formula F contains a unit clause of literal l, then l must be assigned to 1 (true) in order for F to be satisfied. This leads to the unit clause simplification rules [Bre17]:

   i Every clause (other than the unit clause itself) containing l is removed.

   ii If $\neg l$ is contained in any clause then it is removed from the clause. If the clause is then empty, then F is unsatisfiable.

   These simplifications are sound since

   i If l is true then any clause containing l is necessarily true.

   ii If l is true then $\neg l$ is false and so does not positively contribute to the satisfiability of any clause containing $\neg l$.

   For example, consider the formula $F = (a \vee b \vee \neg c) \wedge (a \vee c) \wedge (\neg b \vee \neg d) \wedge b$. The literal b is a unit clause in F, so the clause $(a \vee b \vee \neg c)$ is removed by (i), and $(\neg b \vee \neg d)$ is transformed to $\neg d$ by (ii). So, after unit clause simplification F is transformed to F' where $F' = (a \vee c) \wedge \neg d \wedge b$.

2. *Describe the pure literal rule (PL). Give an example of a* SAT *instance where PL can be applied.*

   A literal is pure if it only occurs as either a positive literal or a negative literal [LLMS10]. For example, in the formula $(a \vee b) \wedge (\neg b \vee \neg c)$, we can see that:

   - a is a pure literal as it only occurs in the form of a positive literal
   - c is also a pure literal as it only occurs as a negative literal
   - b is not a pure literal as it occurs in the formula in both forms

   The pure literal rule entails assigning variables in order to ensure such literals are true, and consequently removing any clauses that contain a pure literal [LLMS10]. It is clear that this simplification rule is sound as not choosing this value for any pure literal l results in l not positively contributing to the satisfiability of any clause.

   As an example, consider the formula $F = (\neg a \vee b) \wedge (c \vee \neg b) \wedge (d \vee \neg e) \wedge (e \vee \neg d)$. We see that a and c are pure literals (as they only occur as negative and positive literals respectively) so any clauses containing them are removed. Thus F is simplified to F' where $F' = (d \vee \neg e) \wedge (e \vee \neg d)$.

3. *Describe the concept of random restarts.*

   The concept of random restarts is very widely used in SAT solving. The idea is to restart the solver after a certain number of clauses have been learnt. The clauses learnt are retained, and the algorithm begins anew from decision level zero but branches differently [GKSS08].

The search may be changed by introducing randomness in to variable selection, or changing the input formula as a result of learnt clauses, or both. In the case of randomness in variable selection, a restart would involve altering a global initialization such as a random seed [RS08].

In general, for a solver and any particular instance, the distribution of running times over different initializations has a very large variance. Random restarts try to take advantage of this by preventing too much time being spent on 'slow' initializations, i.e., initializations in which a satisfying assignment is hard to find. When a restart occurs a new initialization is randomly chosen in the hope that it will lead to a smaller search path. This technique can significantly reduce running time as there can be an exponential difference in running time between 'fast' and 'slow' initializations [RS08].

4. *Describe two variable selection heuristics, for choosing a variable to branch on (and maybe to decide which value we assign to the variable in the first branch).*

   We describe both the Jeroslow-Wang heuristic and the Dynamic Largest Individual Sum (DLIS) heuristic below.

   First, we define the Jeroslow-Wang heuristic. For a formula F in CNF, the Jeroslow-Wang heuristic involves computing

   $$J(l) = \sum_{\omega \in F, l \in \omega} 2^{-|\omega|}$$

   for each literal l. Here $l \in \omega$ means the literal l is present in clause ω, and the length of a clause ω is denoted $|\omega|$.

   A literal l that is unassigned ($l = v$ or $l = \neg v$, where v is unassigned) and with maximal $J(l)$ is chosen to branch on. This heuristic prioritises literals that appear often and in short clauses. Note that the J function need only be computed once, at the start of the solver [KS08].

   We now describe the Dynamic Largest Individual Sum (DLIS) heuristic. This heuristic chooses an unassigned literal that would satisfy the maximum number of unsatisfied clauses. Note that, unlike the Jeroslow-Wang heuristic, this needs to be computed dynamically which introduces significant overhead. Typically some preprocessing is performed in order to compute this more efficiently. Specifically, this can be done by constructing an associated list of clauses C_l for each literal l during the preprocessing step. Then the number of unsatisfied clauses that l would satisfy is calculated dynamically by counting the number of clauses in C_l that are currently unsatisfied [KS08].

5. *Describe Conflict-Driven Clause Learning (CDCL), including any other needed concepts, such as implication graphs. Give an example of a* SAT *instance and an execution of a CDCL algorithm where CDCL helps to decrease the size of the search tree.*

   We will first outline some base concepts before we can fully describe the CDCL algorithm.

   Let us first define the resolution rule. Note that the conjunction of two disjunctive clauses $C_1 \vee a$ and $C_2 \vee \neg a$ (where both C_1 and C_2 are disjunctive clauses) can be resolved to new clause $C_1 \vee C_2$; that is, $(C_1 \vee a) \wedge (C_2 \vee \neg a) \rightarrow C_1 \vee C_2$. This is the resolution rule [MSJ13].

   We will now describe the implication graph. This is a directed graph $G(V, E)$ where each vertex $v \in V$ represents the truth status of a boolean literal and there exists a directed edge $e \in E$ with starting point $a \in V$ and endpoint $b \in V$ if and only if $a \rightarrow b$ (i.e. if and only if a is true implies that b is true) [MSJ13].

   We can now describe the base of the CDCL algorithm; the DPLL (Davis, Putnam, Logemann, Loveland) algorithm. This algorithm works as detailed below [MSJ13]:

```
function DPLLSatisfiabilitySolver(X, F):
    doUnitPropagation(X, F)

    if conflict exists:
        return False

    if all variables assigned:
        return True
```

```
x <- choose variable from X
for value in [True, False]:
    assign value to x
    if DPLLSatisfiabilitySolver(X, F):
        return True

return False
```

Note that unit propagation is as described earlier and can be discovered from the implication graph.

A conflict is detected by the DPLL algorithm whenever we have $a \wedge \neg a$ for any variable a. Let us illustrate this with an example. Consider the following example where $F = (a \vee b) \wedge (\neg a \vee b) \wedge (a \vee \neg b) \wedge (\neg a \vee \neg b)$. Note that we cannot apply the unit clause simplification rule and thus must choose and assign a value to a variable. Let us choose variable a and set its value to false. In the implication graph, we will have an edge from $\neg a$ to b (implied by conjunctive clause $a \vee b$) and thus we must set b to true. However, there will also be an edge from b to a and thus we must set the value of a to true. Since this is a conflict, we must backtrack and try a new value for one of the ancestor nodes in the branching tree.

A CDCL SAT Solver extends the DPLL algorithm with clause learning and non-chronological backtracking, and typically also random restarts, conflict guided branching and lazy data structures [MSJ13].

Learnt clauses result from selected resolution operations being applied when the algorithm arrives at a conflict. Consider the example where we have $F = (\neg a \vee \neg b) \wedge (\neg d \vee b) \wedge (\neg c \vee \neg d \vee a) \wedge (a \vee b \vee c)$ with the following iterations of the algorithm:

(a) First iteration: choose variable c and assign value to true. Nothing further to imply, move to next iteration.

(b) Second iteration: choose variable d and assign value to true. From here we can imply that both a and b must be true. However b true implies a and thus we have a conflict.

From here, we can apply the resolution rule to the involved clauses to obtain a new clause. Note that the involved clauses are $(\neg a \vee \neg b)$, $(\neg d \vee b)$ and $(\neg c \vee \neg d \vee a)$. Applying the resolution rule to $(\neg a \vee \neg b)$ and $(\neg d \vee b)$ we have $(\neg a \vee \neg d)$. Applying it again to $(\neg c \vee \neg d \vee a)$ we have $\neg c \vee \neg d \vee \neg d = \neg c \vee \neg d$. Thus we have a new resolution rule $C_{new} = \neg c \vee \neg d$. We can now return to the first iteration (non-chronological backtracking to one of the causes of the conflict), and include this learnt clause in the original formula F. We set $F' = F \wedge C_{new}$, apply unit propagation and continue the algorithm from the first iteration with F' substituted for F. We can see that by learning this new clause and incorporating it into F we can make sure not to arrive at the same conflict again, thus reducing the size of the search tree.

A problem with clause learning is that it can lead to very large clauses (in the worst case $\mathcal{O}(n)$). A solution to this is what is commonly referred to as *watched literals* [MSJ13]. A naïve way of representing clauses is using adjacency lists, where each variable x keeps a reference to all the clauses that contain the literal x or $\neg x$. This leads to a total number of references L, where L is the number of literals. Watched literals is an example of a lazy data structure, where only two variables per clause keep a reference to it [LLMS10][MSJ13]. We say that only two literals are *watched.* This instead leads to a total number of references $2C$, where C is the number of clauses. In general we have $L \gg C$, and so this representation conserves considerably more space. Using adjacency lists, when a variable is assigned a value, every clause that contains a literal in this variable must be evaluated. With watched literals, only the clauses where literals in the variable are watched need to be evaluated. The idea stems from the fact that unit propagation will only fire when all but one literal in a clause are assigned false, and so there is no need to check a clause until at least one of the watched literals are assigned.

Implementation, Experiments and Interpretation of Results

Implementation and Experiments

Our solver is based on the publicly available SAT solver *pysat* [pys], written in Python. The solver was expanded upon and modified to include random restarts, variable selection heuristics, the pure literal simplification rule, as well as settings to allow varying amounts of retained learnt clauses.

Benchmark instances were taken from SATLIB [sat]. All tests were ran on a set of 40 benchmark instances, distributed evenly over examples with 20, 50, 75 and 100 variables. The solver was timed out if no solution was found after 5 minutes. No memory limits were imposed as memory was never a bottleneck. For all experiments, search restarts were performed after 1000 encountered conflicts. The solver would then restart from decision level 0, retaining all learnt clauses but branching on the next best variable assignment according to the specified variable selection heuristic.

Results

1. *How much does conflict-driven clause learning help in Sat solving? Compare no CDCL, a small amount of CDCL, and a large amount of CDCL.*

   In order to vary the amount of CDCL used, our solver takes a parameter *--max-learnt-clause-length*. Any potential learnt clauses that are longer than this length will not be remembered. These parameter values were 1000, 5, and 0; for a large amount, small amount, and no CDCL (respectively).

   Our results clearly show the beneficial impact of CDCL on running times. The more CDCL that is used, the better the average running time and the fewer time-outs. This is expected since learnt clauses allow the solver to effectively prune large sections of the search space by remembering the causes of past conflicts.

Average Running Time (s)[1]

Amount of CDCL	Number of Variables			
	20	50	75	100
Maximum	0.059	0.147	0.786	2.796
Some	0.056	0.500	25.432	161.866
None	1.894	12.563	171.652	198.230

Number of Time-outs

Amount of CDCL	Number of Variables			
	20	50	75	100
Maximum	0	0	0	0
Some	0	0	0	5
None	0	0	4	6

2. *How much do the simplification rules UP and PL help? Compare only UP, only PL, simplification rules only at every kth level of the search tree (for $k = 2$ or $k = 3$), or an exhaustive use of simplification rules.*

   Our results show that that UP significantly helps improve the running time of the solver. On the other hand, PL seems to have negligible impact. We expected UP to have a large impact, since it is a critical part of the CDCL algorithm. It prevents ever having to branch on units which could otherwise exponentially increase the search space. We found the non-impact of PL to be more surprising. On further research, we found that this characteristic applies to CDCL solvers more generally [ZKF16].

[1] For the purpose of average running time, each timeout was considered to contribute 300 seconds (5 minutes)

Average Running Time (s)[2]

Simplification rules	Number of Variables			
	20	50	75	100
Exhaustive	0.061	0.155	0.807	2.619
Every 2nd level	15.758	118.800	187.274	254.088
Only PL	7.697	150.793	212.522	270.158
Only UP	0.054	0.147	0.875	2.549

Number of Time-outs

Simplification Rules	Number of Variables			
	20	50	75	100
Exhaustive	0	0	0	0
Every 2nd level	0	3	6	8
Only PL	0	5	7	9
Only UP	0	0	0	0

3. *Which variable selection heuristic performs better?*

   We implemented both the Jeroslow-Wang (JW) heuristic and the Dynamic Largest Individual Sum (DLIS) heuristic as described above. For this test, neither heuristic caused the solver to time-out on any test instance.

   Our results indicate the DLIS heuristic outperforms the JW heuristic, especially as the number of variables increases. This is despite DLIS having more overhead, in particular requiring a substantial amount of dynamic computation (i.e., at each branch).

Average Running Time (s)

Variable Heuristic	Number of Variables			
	20	50	75	100
JW	0.056	0.161	0.832	4.500
DLIS	0.060	0.145	0.788	2.803

References

[Bre17] David Bremner. Unit propagation. http://www.cs.unb.ca/~bremner/teaching/cs3383/lectures/unit_propagation/, 2015 (accessed August 22, 2017).

[GKSS08] Carla P. Gomes, Henry Kautz, Ashish Sabharwal, and Bart Selman. Handbook of knowledge representation, 2008.

[KS08] Daniel Kroening and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer, 2008. pp. 39,40.

[LLMS10] J. Larrosa, I. Lynce, and J. Marques-Silva. Satisfiability: Algorithms, applications and extensions. http://sat.inesc-id.pt/~ines/sac10.pdf, 2010.

[MSJ13] J. Marques-Silva and M. Janota. Cdcl sat solvers & sat-based problem solving. http://satsmt2013.ics.aalto.fi/slides/Marques-Silva.pdf, 2013.

[pys] Pysat. https://github.com/cocuh/pysat.

[RS08] V. Ryvchin and O. Strichman. Local restarts in sat. Constraint Programming Letters 4, 06/2008.

[sat] Satlib. http://www.cs.ubc.ca/~hoos/SATLIB/benchm.html.

[ZKF16] Aolong Zha, Miyuki Koshimura, and Hiroshi Fujita. Introducing pure literal elimination into cdcl algorithm. 2016.

[2]See footnote 1