Automatic Software Verification **Spring Semester, 2015**

Lecture 2: March 20, 2015

Lecturer: Prof. Mooly Sagiv *Scribe: Kalev Alpernas and Elizabeth Firman*

2.1 Introduction

In this lecture we will go over the subject of *Satisfiability of Propositional Formulas*.

The Satisfiability Problem

Given a boolean formula in propositional logic, we would like to determine whether the formula:

- is *valid*
  i.e. do all possible assignments satisfy the formula.
  In case the formula is not valid, we sometimes would also like to find a counterexample - a non-satisfying assignment.

- is *satisfiable*
  i.e. there exists an assignment which satisfies the formula.
  In case there exists a satisfying assignment, we sometimes would also like to find that assignment.

As we saw in last week's lecture, we use the result of this problem to verify program. We write the problem, and the negation of the invariant we would like to check as propositional formulas, and check whether they are satisfiable. If there exists a satisfying assignment, then we have found a bug in the program. Otherwise, we have proved that the invariant holds.

Our input formula has 2^n possible assignments, which means that exhaustively checking all assignments is not a feasible approach. In this lecture we will see a few different approaches to solving this problem.

Example 1. We consider for example the following propositional formula:

$$\varphi = (a \vee b) \wedge (\neg a \vee \neg b \vee c)$$

In Figure 2.1 we can see a tree representing the truth state of φ. The colour Green represents a satisfying assignment, and Red represents a non-satisfying assignment. We can see that φ is not valid, for example since the assignment $a = F, b = F, c = T$ is not a satisfying assignment. We can also see that the formula is satisfiable, since the assignment $a = F, b = T, c = T$ is satisfying.

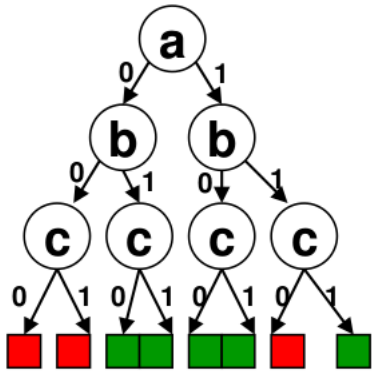

Figure 2.1: The truth value search tree for φ

Problem Representation

While the input for the SAT problem is a propositional formula, we will work with formulas in *Conjunctive Normal Form* or *CNF*. A CNF formula is a conjunction of *clauses*, where each clause is a disjunction of *Literals*. A literal is a boolean variable, or its negation.

We choose to represent the problem in CNF, as it allows us to represent the formula in a compact, simple data structure, it is compositional, and in addition the fact that each clause in the formula needs to be satisfied, lends itself to algorithms which are easier to grasp intuitively.

Any propositional formula can be converted to CNF, and the conversion is a straightforward process, utilizing basic logic equivalences - De-Morgan, double negation, and distributivity. This process may cause the formula to grow, so in order to avoid a size explosion of the formula we might be required to add additional variables to the formula.

Example 2. We consider the following conversion of the formula $\varphi = a \vee (b \wedge \neg(c \vee \neg d))$ to CNF:

$$\begin{aligned} \varphi = a \vee (b \wedge \neg(c \vee \neg d)) \equiv (a \vee (b \wedge \neg c \wedge \neg\neg d)) \equiv \\ (a \vee (b \wedge \neg c \wedge d)) \equiv (a \vee b) \wedge (a \vee \neg c) \wedge (a \vee d) \end{aligned} \tag{2.1}$$

Complexity Results

In [1] Cook showed that the Boolean Satisfiability Problem is NP-Complete, even if we limit the clause size to at most 3 literals[1].

Two noteworthy exceptions are *2-SAT*, and *Horn Satisfiability*. These two problems have polynomial algorithms (in [3] and [4] respectively).

[1] Known as the 3-SAT problem.

2-SAT is the Boolean satisfiability problem where each clause is limited to at most 2 literals.

Horn Satisfiability is the Boolean satisfiability problem where each clause has at most one positive literal[2].

What is it Good For?

From the perspective of complexity analysis, the boolean satisfiability problem is a very useful tool for showing complexity results of many problems in CS.

In [2] Karp uses the complexity result of [1] to show that 21 other problems[3] are also NP-Complete, by showing a reduction from the Boolean Satisfiability problem to each of these problems. Many similar results have been shown for further problems since.

From the perspective of verification, SAT solvers have proved to be very useful tools in solving problems in SW and HW verification, as well as in other fields such as AI. This is because despite the problem being NP-Complete in the worst case, there are many algorithms which are capable of solving the Boolean Satisfiability problem in many useful cases.

This lecture will show us a few such algorithms, which solve the SAT problem with exponential worst case complexity, but work well on many useful instances.

2.2 Resolution

The first approach to solving the SAT problem that we will see in this lecture is the *Clause Resolution* method.

Clause Resolution

Resolution of a pair of clauses with one incompatible variable, is a process where given two clauses, in which there is exactly one variable that has incompatible truth values, i.e. the variable appears in one clause, and its negation appears in the other, we derive a new clause, called the *resolvent*, which is the disjunction of the original two clauses, without the incompatible variable.

Example 3. In Figure 2.2 we can see the resolution method applied to the clauses $(a \vee b \vee \neg c \vee f)$ and $(g \vee \neg h \vee c \vee f)$.

The incompatible variable is c, and we can see that the resulting clause is a disjunction of $(a \vee b \vee f)$ and $(g \vee \neg h \vee f)$ (f appears in both original clauses, so it only appears once in the resolvent).

The following deduction is the formal justification for single variable applying resolution:

$$\frac{A \implies B \qquad \neg A \implies C}{B \vee C}$$

[2]Known as Horn clause.

[3]Known as Karp's 21 NP-Complete problems.

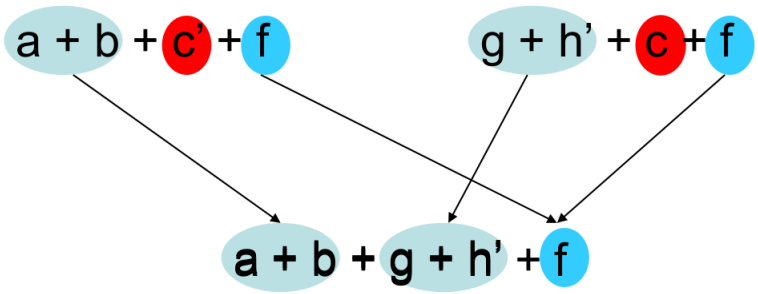

Figure 2.2: An example of resolution on the variable c

Note that every clause is equivalent to an implication formula where the rhs of the implication is a disjunction of literals.

Claim 1. *The resolvent and the original clauses are* equisatisfiable*, i.e. the original clauses are satisfiable iff the resolvent is satisfiable. Furthermore, the same assignment satisfies both.*

Proof Sketch: Assume that some model M satisfies the initial two clauses, and we are resolving the variable a. We assume w.l.o.g that a holds in M. Hence, at least one of the literals in the clause that contains $\neg a$ has to hold in M. The same literal will appear in the resolvant. We get that if the original clauses were satisfiable, the resolvant will be satisfiable as well. ■

From Claim 1 we get that applying resolution on two clauses in a CNF formula preserves the satisfiability of the formula. In a similar sense we get that resolution preserves the validity of a formula, since validity of a formula is equivalent to the satisfiability of it's negation, and satisfiability is preserved under resolution.

Davis-Putnam Algorithm

In [5] Davis and Putnam proposed an algorithm for solving the SAT problem. The algorithm is shown in Figure 2.3.

If the algorithm reaches an empty clause as a resolvent, then there is no satisfying assignment for the original formula, and we can say that the formula is not satisfiable (let alone valid).

If, however, the algorithm terminated without reaching an empty clause, then the formula is satisfiable, and there are no incompatible variables remaining in any two clauses. Finding a satisfying assignment at this point is straightforward.

Example 4. In Figure 2.4 we see the application of the resolution method on the formula $\varphi = (a \vee b \vee c) \wedge (b \vee \neg c \vee \neg f) \wedge (\neg b \vee e)$. In the first iteration the

for all Variables in formula **do**
  Resolve all clauses for variable
  if Reached empty clause **then**
    Report UNSAT
  else
    Discard resolved clauses
  end if
end for

Figure 2.3: The Davis-Putnam Algorithm

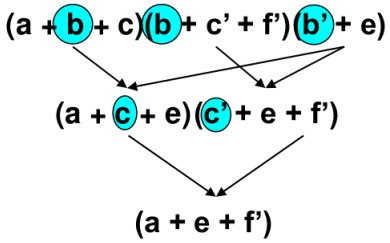

Figure 2.4: An example of resolution of the formula $\varphi = (a \vee b \vee c) \wedge (b \vee \neg c \vee \neg f) \wedge (\neg b \vee e)$

variable b is chosen, and two resolvents are produced. In the second iteration the variable c is chosen and a single resolvent is produced.

The algorithm terminated without reaching an empty clause, which means that this formula is satisfiable.

Example 5. In Figure 2.5 we see the application of the resolution method on the formula $\varphi = (a \vee b) \wedge (a \vee \neg b) \wedge (\neg a \vee c) \wedge (\neg a \vee \neg c)$. In the first iteration the variable b is chosen, one resolvent is produced and two other clauses are carried without change. In the second iteration the variable a is chosen and two resolvents are produced. In the third iteration the variable c is chosen, and the empty clause is produced.

The algorithm terminated after reaching an empty clause, which means that this formula is not satisfiable.

This algorithm may potentially produce an exponential amount of clauses, since in the worst case in every iteration every pair of clauses is resolved, resulting in an exponential space complexity.

Example 6. Consider the following formula:

$$\varphi = (a \vee b_1) \wedge (a \vee b_2) \wedge ... \wedge (a \vee b_n) \wedge (\neg a \vee c_1) \wedge ... \wedge (\neg a \vee c_n)$$

After resolving all clauses for a we get:

$$\varphi^1 = (b_1 \vee c_1) \wedge ... \wedge (b_1 \vee c_n) \wedge ... \wedge (b_n \vee c_1) \wedge ... \wedge (b_n \vee c_n)$$

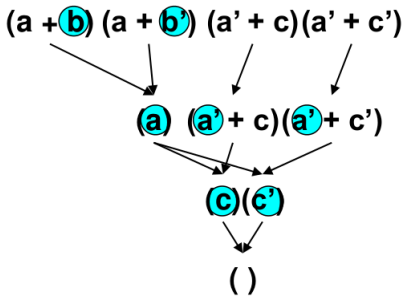

Figure 2.5: An example of resolution of the formula $\varphi = (a \vee b) \wedge (a \vee \neg b) \wedge (\neg a \vee c) \wedge (\neg a \vee \neg c)$

φ had $2n$ clauses, whereas in φ^1 has n^2 clauses. After i iterations we would have $2\frac{n^{j-1}}{2^j}$, i.e. an exponential blow-up.

A point of note is that in case the formula is satisfiable, the order of the variables selected may greatly affect the outcome of the algorithm. If the formula is not satisfiable, then the algorithm is guaranteed to reach an empty clause regardless of the order of the variables.

2.3 DPLL

The DLL [6] algorithm, also known as DPLL, is a refinement of the Davis-Putnam algorithm. It's the basic framework of many existing SAT solvers.

2.3.1 The Basic DPLL Algorithm

To avoid using exponential space we don't want to apply resolution on all the clauses, but on some reduced set. We can do it in a very straightforward way: Given a set of clauses we do a Depth First Search on the decision tree (a reduced version). For some order of variables, we assign 0 to the first literal (according to the order of the DFS scan) and remove all the satisfied clauses. Then we have a reduced set of clauses with a reduced set of literals (the ones that weren't assigned yet). We recursively keep assigning values until we reach a satisfiable assignment or a contradiction. Also, at some point we would like to apply resolution instead of keep assigning values. We want to stop assigning as early as we can, but we want to do it without the exhaustive cost of applying resolution.

Before introducing the DPLL algorithm lets go over some key definitions that the algorithm uses:

Implication: A variable is forced to be assigned to be True or False based on previous assignments.

Unit Clause: An unsatisfied clause that has exactly one unassigned literal (all the other literals were already set).

Unit Clause Rule: The unassigned literal in the unit clause is implied.

Boolean Constraint Propagation (BCP): Iteratively apply the unit clause rule until there are no unit clauses available.

The DPLL algorithm returns SAT if the formula is satisfiable, UNSAT otherwise. We define the algorithm as follows:

while true **do**
  if If all variables are assigned **then**
    Return SAT
  end if
  Choose the next variable and value
  while Apply BCP and a conflict is reached **do**
    Resolve conflict - backtrack until there's no conflict
    if Conflict cannot be resolved **then**
      Report UNSAT
    end if
  end while
end while

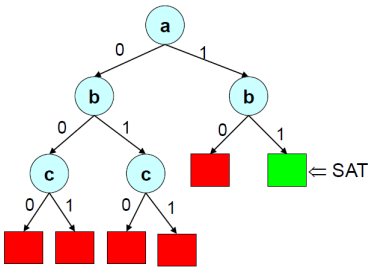

Figure 2.6: The decision tree of Example 6 with applied BCP

Example 7. Consider the formula $(a^{'}+b+c)(a+c+d)(a+c+d^{'})(a+c^{'}+d)(a+c^{'}+d^{'})(b^{'}+c^{'}+d)(a^{'}+b+c^{'})(a^{'}+b^{'}+c)$ We first try to explore the case of $a=0$, if it won't work we will explore the case of $a=1$. We assign first according to DFS $a=0, b=0, c=0$, until we're left with the formula $(a+c+d)(a+c+d^{'})$. Now we apply resolution, which is cheap since we choose to resolve unit clauses. For each clause we can infer the value of d and reach a contradiction. In Figure 2.7 we can see the implication graph and the reached conflict. Since we have a conflict, we backtrack and check the assignment $c=1$. We call this assignment a forced decision since we exhausted the search for $c=0$. We backtrack again and continue as described in the algorithm. Lets look at the final branch where $a=1, b=1$. Note we got this assignment after backtracking and forced decisions. For this assignment we are left with the formula $(b^{'}+c^{'}+d)(a^{'}+b^{'}+c)$ and apply BCP: First we apply the unit clause rule on $(a^{'}+b^{'}+c)$ and get the assignment $c=1$. Then we apply the unit

clause rule on $(b^{'} + c^{'} + d)$ and get the assignment $d = 1$. See the implication graph in Figure 2.8. We can see that the conflict was finally resolved and we got a satisfying assignment.

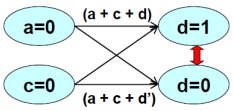

Figure 2.7: in Example 6 a conflict is reached for $a = 0, b = 0, c = 0$

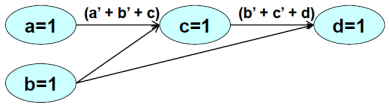

Figure 2.8: The implication graph for the assignment $a = 1, b = 1$ in Example 6

2.3.2 Conflict Driven Learning

In the basic DPLL algorithm we backtracked only one level up, chronologically according to DFS. But we can imply a better heuristic which is not in the original DPLL algorithm. We can achieve non chronological backtracking as a result of the conflict driven learning. In this heuristic we take advantage even of the "unsuccessfull" explorations, by exploiting the variables' assignments that led to a conflict. Meaning, when reaching a conflict, we can remember the variables that led us there, and create a new clause with those variable. This clause is added to the set of clauses, and we backtrack to the lowest level where one of the variables was involved. Note that we can backtrack more than one level up, as opposed to the basic DPLL. By learning the new clause from the conflict, we can limit our explorations - whenever we reach a point in the tree with those assignments, we won't have to continue until a contradiction is reached, since the additional clause will imply that the assignment is not a satisfying one.
Lets look at the following example to better understand how the conflict driven learning heuristic works and how it differs from the basic DPLL.

Example 8. Consider the formula $(x_1 + x_4)(x_1 + x_3^{'} + x_8^{'})(x_1 + x_8 + x_{12})(x_2 + x_{11})(x_7^{'} + x_3^{'} + x_9)(x_7^{'} + x_8 + x_9^{'})(x_7 + x_8 + x_{10}^{'})(x_7 + x_{10} + x_{12}^{'})$.

Lets run he basic DPLL (Figure 2.9): First we assign $x_1 = 0$, $(x_1 + x_4)$ becomes a unit clause so according to the unit clause rule we get $x_4 = 1$. Next we assign $x_3 = 1$ and by the BCP we get $x_8 = 0, x_{12} = 1$. Now we choose again and set $x_2 = 0$, by applying BCP we get $x_{11} = 1$. We choose an assignment again of $x_7 = 1$. By applying BCP we get a contradiction of the clauses $(x_7^{'} + x_3^{'} + x_9)(x_7^{'} + x_8 + x_9^{'})$. It's easy to see that the assignment that caused

this contradiction is $x_3 = 1, x_7 = 1, x_8 = 0$.

Conflict Learning (Figure 2.10): Now we deviate from the basic DPLL, and use the new acquired knowledge from the contradiction. We learned that this specific assignment to these variables leads to a conflict, based on choices we made before. Lets call it a local conflict. We can benefit from remembering local conflicts, since if we reach this assignment again as a result of some other assignments, we can avoid reaching the same conflict. To achieve this goal we create a new clause - a conflict clause, that is evaluated to false given the local conflict assignment. Here the conflict clause is: $x_3^{'} + x_7^{'} + x_8$. Notice that this clause is a result of applying resolution on $(x_7^{'} + x_3^{'} + x_9)(x_7^{'} + x_8 + x_9^{'})$.

Non Chronological Backtracking (Figure 2.11): After adding a new conflict clause, we backtrack to the lowest level that included one of the variables in this clause. Here we backtrack to the decision level $x_3 = 1, x_8 = 0, x_{12} = 1$. In this level we will choose $x_7 = 0$, as a result of applying BCP to the newly added conflict clause.

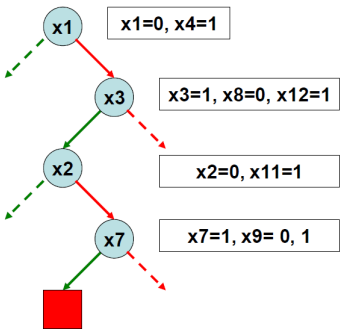

Figure 2.9: The decision tree of Example 7 with applied DPLL

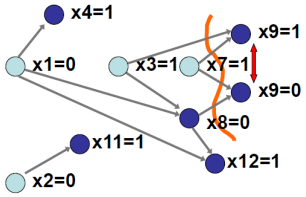

Figure 2.10: In Example 7 we can look at this decision graph, and take the cut that leads to the conflict. We can deduce the conflict clause from the sources of the edges that are in the cut: x_3, x_7, x_8.

Note that by adding clauses we give up the polynomial space complexity of the basic DPLL, and we can reach exponential space complexity in the worst case.

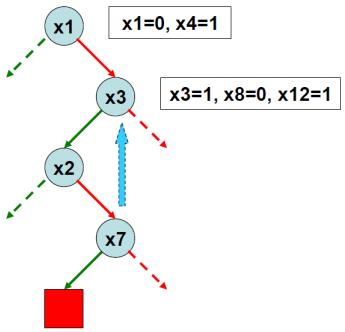

Figure 2.11: In example 7, after adding the conflict clause we backtrack to the level of x_3, which is the lowest level where the variables of the conflict clause participate.

2.3.3 Restart

Another heuristic that some SAT solvers apply is restarting the assignment tree. At a certain point the SAT solver can give up the current tree it uses for its search, and start over. For example decide on a different variable order. But it won't start completely from scratch, the conflict clauses that were learned are part of the clause set. This adds to the robustness of the solver, and can help reducing the search space. Note that every time the SAT solver restarts, it can start over with a different set of clauses. This also contributes to the different assignment the SAT solver can produce.

2.3.4 BCP Algorithm

What we did so far in the BCP step, is iteratively assign literals in unit clauses to their implied value. But there is a more efficient heuristic that minimizes the clause access. Instead of just propagating assignments in unit clauses, we choose 2 literals in each clause and call them the "watch literals". For each clause, if its watch literals remain unassigned in the current iteration we ignore other assignments of variables in the clause and do nothing. Once a watch literal is assigned to False (otherwise the clause is satisfied), we choose the next unassigned variable in the clause to be the watch literal. We keep propagating assignments until the clause becomes a unit clause and the value of the last variable is implied. If there's no unit clause in the current iteration, we choose the next variable and assignment and continue to the next iteration. Note that we don't inspect a clause until one of its watch literals is assigned, as opposed to the naive BCP that inspect clauses with every assignment.

Example 9. Consider the formula $(v_2 + v_3 + v_1 + v_4 + v5)(v_1 + v_2 + v_3^{'})(v_1 + v_2^{'})(v_1^{'} + v_4)(v_1^{'})$. First we will choose the first two literals of every clause (with more than one literal) to be the watch literals of that clause. From the unit clause $(v_1^{'})$ the assignment $v_1 = 0$ is implied. Now we inspect the clauses where

the watched literals were set to false. For the second clause we replace the watched literal v_1 with the unassigned literal v_3' (Figure 2.12), and since the third clause is a unit clause, the value of the second watch literal is implied and we get $v_2 = 0$ (Figure 2.13). Note that this assignment sets the first watched literal in the second clause to False, therefore we examine it and discover we have a unit clause. Once again we get from implication a variable assignment $v_3 = 0$. Since it affects the first watched literal of the first clause, we replace it with v_5 which becomes the second watch literal in the first clause (Figure 2.14). At this point the BCP terminates and we choose the next variable and a value. Lest say the algorithm chose $v_4 = 1$. No watched literals were affected (we ignore the last clause since it's already satisfied), and there's no contradictions. The BCP has nothing to do and we choose the next variable. The algorithm chooses the remaining unassigned variable v_5 and sets it to False. We do not examine the first clause since it's already satisfied, also v_5 is not watched in any other clause. The algorithm terminates with a satisfying assignment (Figure 2.15).

v2 + v3 + v1 + v4 + v5
v1 + v2 + v3′
v1 + v2′
v1′+ v4

Figure 2.12: The watched literals in Example 8, after the first assignment and the examination of the second clause. Here the Red literals are set to False, and the Green literals are set to True.

V2 + v3 + v1 + v4 + v5
v1 + v2 + v3′
v1 + v2′
v1′+ v4

Figure 2.13: The examination of the third clause in Example 8, leads to implication of the second watched literal. The colors are set as in Figure 11.

V2 + v3 + v1 + v4 + v5
v1 + v2 + v3′
v1 + v2′
v1′+ v4

Figure 2.14: The unit clause propagation in Example 8, after we set the value of v_2 to False. Also a new watch literal is selected for the second clause. The colors are set as in Figure 11.

V2 + v3 + v1 + v4 + v5
v1 + v2 + v3′
v1 + v2′
v1′+ v4

Figure 2.15: The final assignment in Example 8. The colors are set as in Figure 11.

2.3.5 Chaff Decision Heuristic

We can see that the DPLL algorithm is very sensitive to the order of variables. We would like to choose the optimal order that will result with the minimal decision tree. We can't find the best order, but we can apply a heuristic that helps us choose the next variable in a smarter way. This heuristic is called Variable State Independent Decaying Sum - VSIDS. The idea is to order the variables according to the literal count in the initial clause set, to increment the counts only when new clauses are added (conflict clauses) and periodically divide all counts by a constant. It's a Quasi-static algorithm, since it's both static and non static. It's static because it doesn't depend on variable state, and it's not static because it gradually changes as new clauses are added.

2.4 Game Theoretic Approach to the SAT Problem

An alternative way to approach finding a solution for the SAT problem is to consider it as a 2 player game. The winning condition for Player 1 is if she finds a satisfying assignment, the goal of Player 2 is to show that no such assignment exists.

The initial state is an empty assignment, and the original formula. At her turn Player 1 updates the assignment such that the new assignment satisfies a clause that was not satisfied by the previous assignment, and Player 2 updates the formula by adding a clause that is implied by the formula and is not satisfied by Player 1's assignment.

Example 10. We consider the example of solving the SAT problem for the formula $\varphi = (a \vee b) \wedge (a \vee \neg b) \wedge (\neg a \vee c) \wedge (\neg a \vee \neg c)$ with a 2-player game:

Initially the assignment is empty, and the formula is φ.

	Player 1 (Assignment)	Player 2 (Formula)
0	$[]$	$(a \vee b) \wedge (a \vee \neg b) \wedge (\neg a \vee c) \wedge (\neg a \vee \neg c)$
1	$[a \mapsto T]$	$(a \vee b) \wedge (a \vee \neg b) \wedge (\neg a \vee c) \wedge (\neg a \vee \neg c) \wedge (\neg a)$
2	$[a \mapsto F]$	$(a \vee b) \wedge (a \vee \neg b) \wedge (\neg a \vee c) \wedge (\neg a \vee \neg c) \wedge (\neg a) \wedge (a)$
3	$[a \mapsto F, b \mapsto T]$	$(a \vee b) \wedge (a \vee \neg b) \wedge (\neg a \vee c) \wedge (\neg a \vee \neg c) \wedge (\neg a) \wedge (a) \wedge ()$

On round 1 Player 1 assigns a to be T. Player 2 counters by deriving the clause $(\neg a)$ which is not satisfied by the assignment. On round 2 Player 1 changes the assignment of a to F so that the clause added in round 1 is no longer contradicted. Player 2 counter by deriving the clause (a). On round 3 Player 1 adds the assignment of b to T to the assignment. Player 2 counters by deriving the empty clause, showing that the formula is not satisfiable, thus winning the game.

.1 Recitation

In the recitation we've seen several examples for using SAT solvers.

Graph Colouring

The first example we've seen was a reduction from the graph colouring problem to the SAT, leveraging the SAT solver to find a solution to the colouring problem.

The input to the graph colouring problem is a graph $G = (V, E)$ and a natural number k, where $V = \{0, 1, ..., n-1\}$ and $E \subseteq V \times V$. The question we would like to answer is whether there exists a colouring $C : V \to \{0, ..., k-1\}$ such that $\forall v_1, v_2 \in V : (v_1, v_2) \in E \Rightarrow C(v_1) \neq C(v_2)$, or intuitively, no two adjacent vertices have the same colour.

Reduction to SAT:

Variables

$\forall v \in V, c \in \{0, ..., k-1\} : b_{v,c}$. Intuitively $b_{v,c}$ means that the variable v is coloured with the colour c, or $b_{v,c} \Leftrightarrow C(v) = c$.

Clauses

$\forall v \in V : b_{v,0} \vee b_{v,1} \vee ... \vee b_{v,k-1}$. A variable has to have at least one colour.

$\forall c_1 \neq c_2 \in \{0, ..., k-1\}, v \in V : \neg b_{v,c_1} \vee \neg b_{v,c_2}$. A variable has to have at most one colour.

$\forall c \in \{0, ..., k-1\}, (v_1, v_2) \in E : \neg b_{v_1,c} \vee \neg b_{v_2,c}$. Adjacent variables can't have the same colour.

Hamiltonian Path

The input to the Hamiltonian path problem is a graph $G = (V, E)$ as before. The question we would like to answer is whether there exists a path which visits every node in the graph exactly once. A path is a series of nodes $(v_1, ..., v_n), v_j \in V$, where $(v_i, v_{i+1}) \in E$.

Reduction to SAT:

Variables

$b_{i,j}$. Intuitively $b_{i,j}$ means that the j-th vertex in the path is the vertex i, or $b_{i,j} \Leftrightarrow v_j = i$.

Clauses

For every item in the path we construct the following clause (presented here for the first path item):
$\forall j_m s.t.(i, j_m) \in E : b_{i,0} \implies (b_{j_1,1} \vee b_{j_2,1} \vee ... \vee b_{j_k,1})$

Note that the actual encoding of the clauses in the code is different than the one presented here.

Unsat Core

In case the formula we are checking is unsatisfiable, we would like, in some cases, to get a reduced formula which is itself unsatisfiable. This is called an *Unsat Core*.

In Z3, to get an Unsat Core, we need to pass a set of additional variables (or formulas) to the call to solver.check(). The solver will then return a subset of these clauses which, along with the other formulas added to the solver, are unsatisfiable.

In the graph colouring example, we added a new variable per vertex in the graph, and disjoined each variable with the clause that constrained the colour of each vertex. We then passed these variables to the check() call. The resulting unsat core is a subgraph of the original graph, which does not have a k colouring.

Note that in Z3 the unsat core is not necessarily minimal, neither locally, i.e. there might be a strict subset of the unsat core which is itself unsatisfiable, nor globally, i.e. there might be smaller, disjoint subset of the problem which was not reported.

Bibliography

[1] S. A. Cook, *The complexity of theorem proving procedures*. Proceedings Third Annual ACM Symp. on the Theory of Computing,1971, 151-158.

[2] R. M. Karp, *Reducibility among combinatorial problems*. Complexity of Computer Computations: Proc. of a Symp. on the Complexity of Computer Computations, 1972, pp. 85-103.

[3] Krom, Melven R., *The Decision Problem for a Class of First-Order Formulas in Which all Disjunctions are Binary*. 1967, Zeitschrift fr Mathematische Logik und Grundlagen der Mathematik 13: 1520.

[4] Dowling, W., and Gallier, J., *Linear-time algorithms for testing the satisfiability of propositional Horn formulae*. 1984, Journal of Logic Programming, 3, 267-284

[5] M .Davis, H. Putnam, *A computing procedure for quantification theory*. 1960, J. of ACM, Vol. 7, pp. 201-214

[6] M. Davis, G. Logemann and D. Loveland, *A Machine Program for Theorem-Proving*. 1962, Communications of ACM, Vol. 5, No. 7, pp. 394-397