Proving SAT Solving Algorithms and Data Structures in Dafny

Proposed by ANDRICI Cezar Constantin
Adviser Assoc. Prof. Stefan Ciobaca, PhD

Boolean Satisfiability Problem (SAT)

Given a boolean formula, find a truth assignment that evaluates the formula to true. If such truth assignment does not exists, then return that the formula is not satisfiable.

The SAT problem can solve many practical applications such as electronic design automation, software and hardware verification, artificial intelligence, and operations research.

Modern SAT Solvers can solve tests with millions of variables in reasonable time.

Objective:

Construct a competitive, formally verified implementation of a SAT solving algorithm

Dafny

```
method BinarySearch(a: array<int>, key: int) returns (r: int)
  requires ∀ i,j • 0 ≤ i < j < a.Length ⟹ a[i] ≤ a[j]
  ensures 0 ≤ r ⟹ r < a.Length ∧ a[r] = key
  ensures r < 0 ⟹ key ∉ a[..]
{
  var lo, hi := 0, a.Length;
  while lo < hi
    invariant 0 ≤ lo ≤ hi ≤ a.Length
    invariant key ∉ a[..lo] ∧ key ∉ a[hi..]
  {
    var mid := (lo + hi) / 2;
    if key < a[mid] {
      hi := mid;
    } else if a[mid] < key {
      lo := mid + 1;
    } else {
      return mid;
    }
  }
  return -1;
}
```

Proved Algorithms

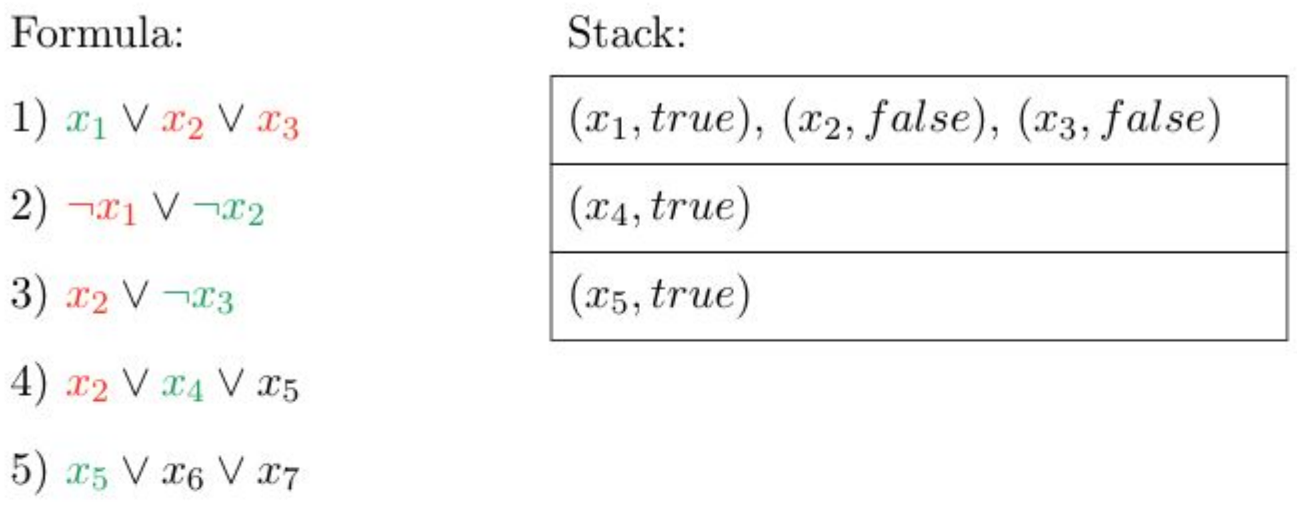
Basic Backtrack Search

- Implemented using pure functions;
- Inefficient because will go through all the search space;
- We used this as as a chance to accommodate with the problem, Dafny and the pre- and post-conditions necessary for proving the correctness of the solution returned by the program.

Davis-Putnam-Logemann-Loveland Algorithm

- Implemented using imperative version;
- prunes the search space by avoiding falsified clause;

Unit Propagation



Formula:

1) $x_1 \lor x_2 \lor x_3$

2) $\neg x_1 \lor \neg x_2$

3) $x_2 \lor \neg x_3$

4) $x_2 \lor x_4 \lor x_5$

5) $x_5 \lor x_6 \lor x_7$

Stack:

$(x_1, true)$, $(x_2, false)$, $(x_3, false)$
$(x_4, true)$
$(x_5, true)$

Figure 1: Stack representation for example 1

Data Structure 1: Stack

```
class Stack {
    var size : int;
    var stack : array< seq<(int, bool)> >;
    var variablesCount : int;
    ghost var contents : set<(int, bool)>;
}
```

Data Structure 2: Formula

```
class Formula {
    var variablesCount : int;
    var clauses : seq< seq<int> >;
    var stack : Stack;

    var truthAssignment : array<int>; // from 0 to variablesCount - 1, values: -1, 0, 1

    var trueLiteralsCount : array<int>; // from 0 to |clauses| - 1
    var falseLiteralsCount : array<int>; // from 0 to |clauses| - 1

    var positiveLiteralsToClauses : array< seq<int> >; // from 0 to variablesCount - 1
    var negativeLiteralsToClauses : array< seq<int> >; // from 0 to variablesCount - 1
}
```

Predicates

Invariant 2. A variable appears only once in the stack.

```
∀ i,j • 0 ≤ i < |stack| ∧  0 ≤ j < |stack[i]| ⟹
    (∀ i', j' • i < i' < |stack| ∧ 0 ≤ j' < |stack[i']| ⟹
        stack[i][j].0 ≠ stack[i'][j'].0)
    (∀ j' • j < j' < |stack[i]| ⟹
        stack[i][j].0 ≠ stack[i][j'].0)
  )
```

Invariant 4. *validTruthAssignment*()

```
|truthAssignment| = variablesCount ∧

(∀ i • 0 ≤ i < |truthAssignment| ⟹ -1 ≤ truthAssignment[i] ≤ 1)

∧

(∀ i • 0 ≤ i < |truthAssignment| ∧ truthAssignment[i] ≠ -1 ⟹
  (i, truthAssignment[i]) in stack.contents)

∧

(∀ i • 0 ≤ i < |truthAssignment| ∧ truthAssignment[i] = -1 ⟹
  (i, false) ∉ stack.contents ∧ (i, true) ∉ stack.contents)
```

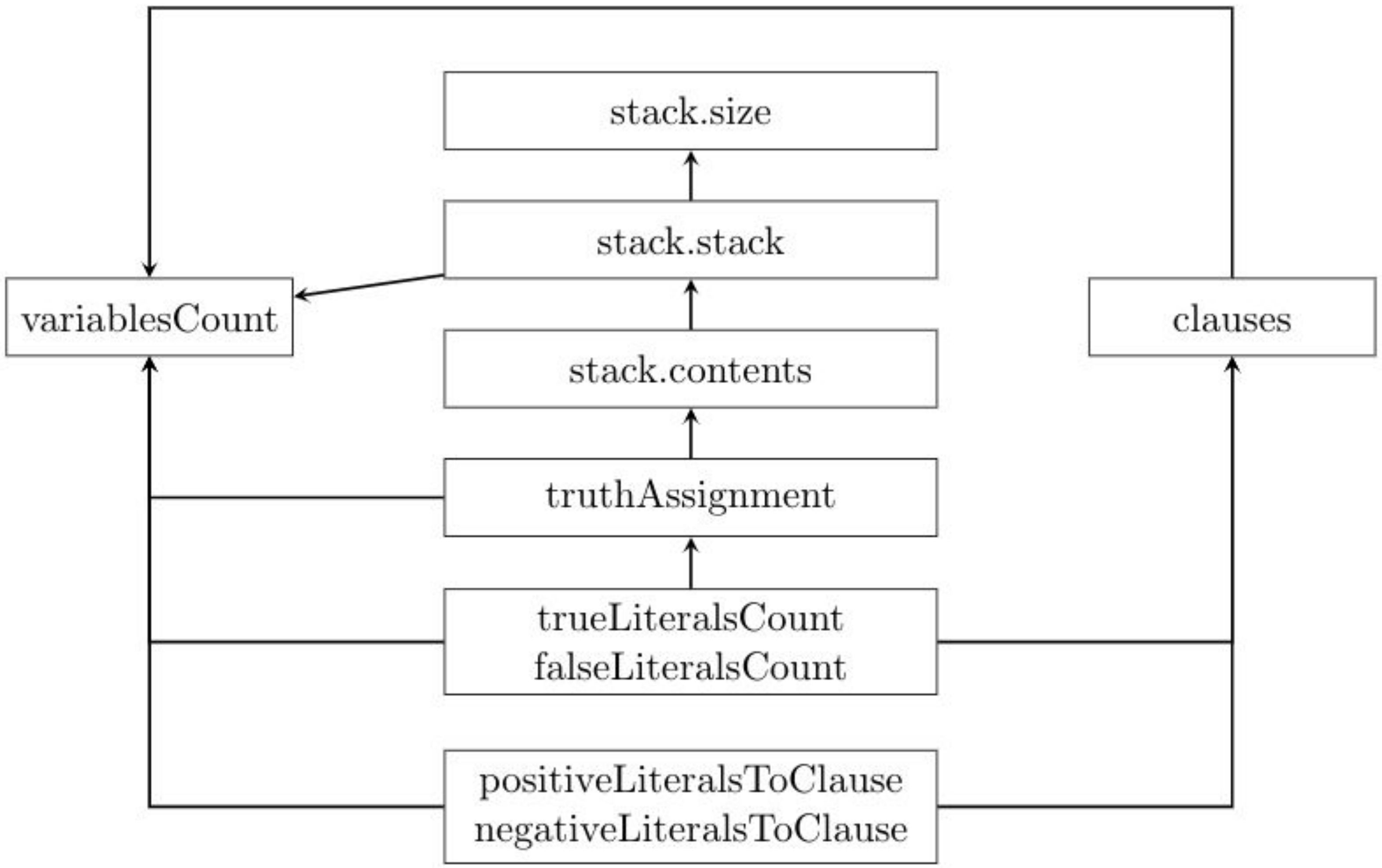


Figure 2: The dependencies between the data structures

```
method solve() returns (result : SAT_UNSAT)
    ...
    ensures result.SAT? ⟹ formula.isSatisfiableExtend(formula.truthAssignment);
    ensures result.UNSAT? ⟹
      ¬formula.isSatisfiableExtend(formula.truthAssignment);
```

Predicate 5. $isSatisfiableExtend(tau, clauses)$: A set of clauses are satisfiable extend by a truth assignment tau if
$\exists tau' \bullet isTauComplete(tau') \land isExtendingTau(tau, tau') \land isSatisfied(tau')$.

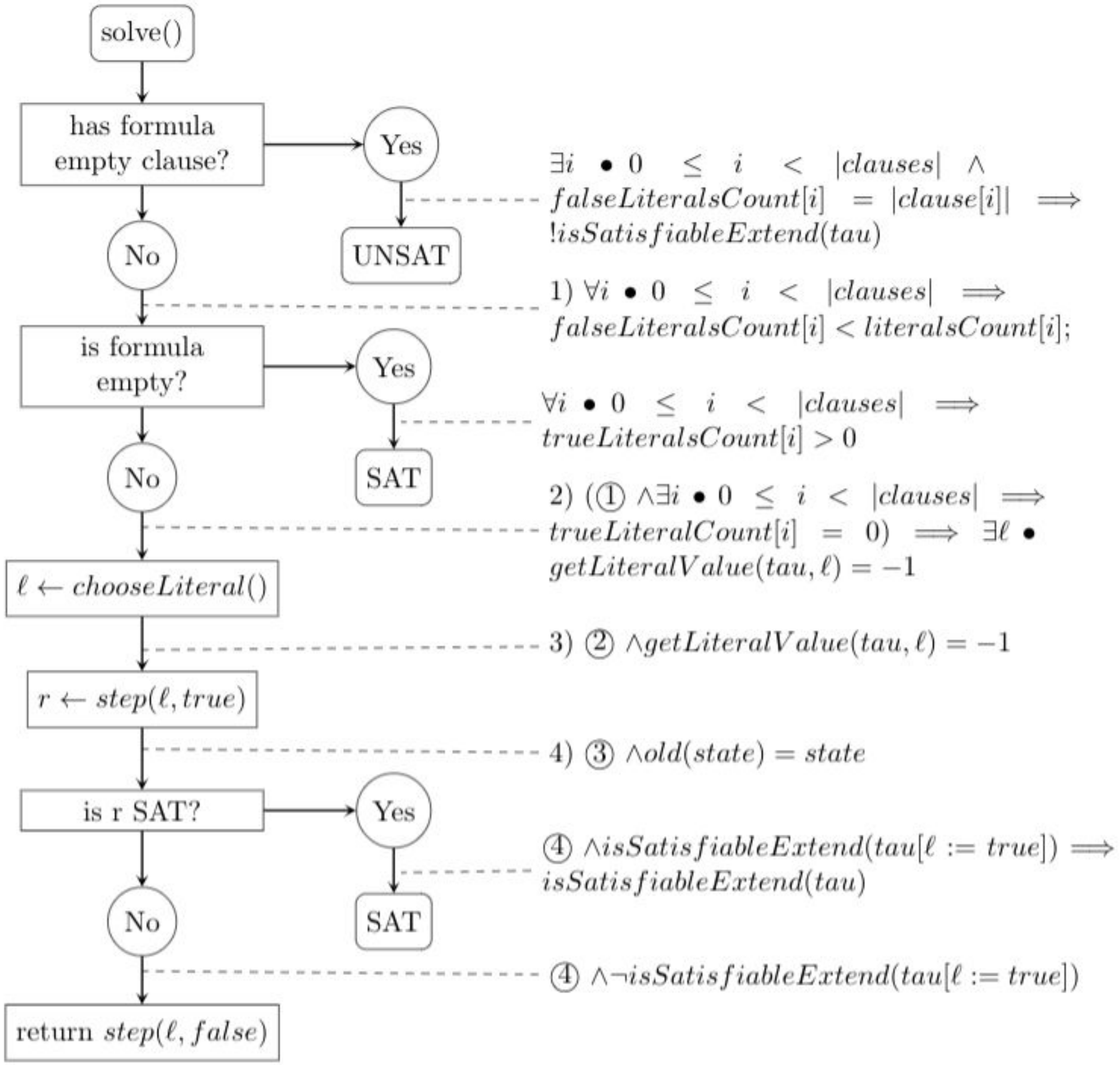

solve()
has formula empty clause?
Yes
UNSAT
$\exists i \bullet 0 \leq i < |clauses| \wedge falseLiteralsCount[i] = |clause[i]| \implies !isSatisfiableExtend(tau)$
No
1) $\forall i \bullet 0 \leq i < |clauses| \implies falseLiteralsCount[i] < literalsCount[i];$
is formula empty?
Yes
SAT
$\forall i \bullet 0 \leq i < |clauses| \implies trueLiteralsCount[i] > 0$
No
2) $(① \wedge \exists i \bullet 0 \leq i < |clauses| \implies trueLiteralCount[i] = 0) \implies \exists \ell \bullet getLiteralValue(tau, \ell) = -1$
$\ell \leftarrow chooseLiteral()$
3) $② \wedge getLiteralValue(tau, \ell) = -1$
$r \leftarrow step(\ell, true)$
4) $③ \wedge old(state) = state$
is r SAT?
Yes
SAT
$④ \wedge isSatisfiableExtend(tau[\ell := true]) \implies isSatisfiableExtend(tau)$
No
$④ \wedge \neg isSatisfiableExtend(tau[\ell := true])$
return $step(\ell, false)$

Figure 3: Flowchart of method solve

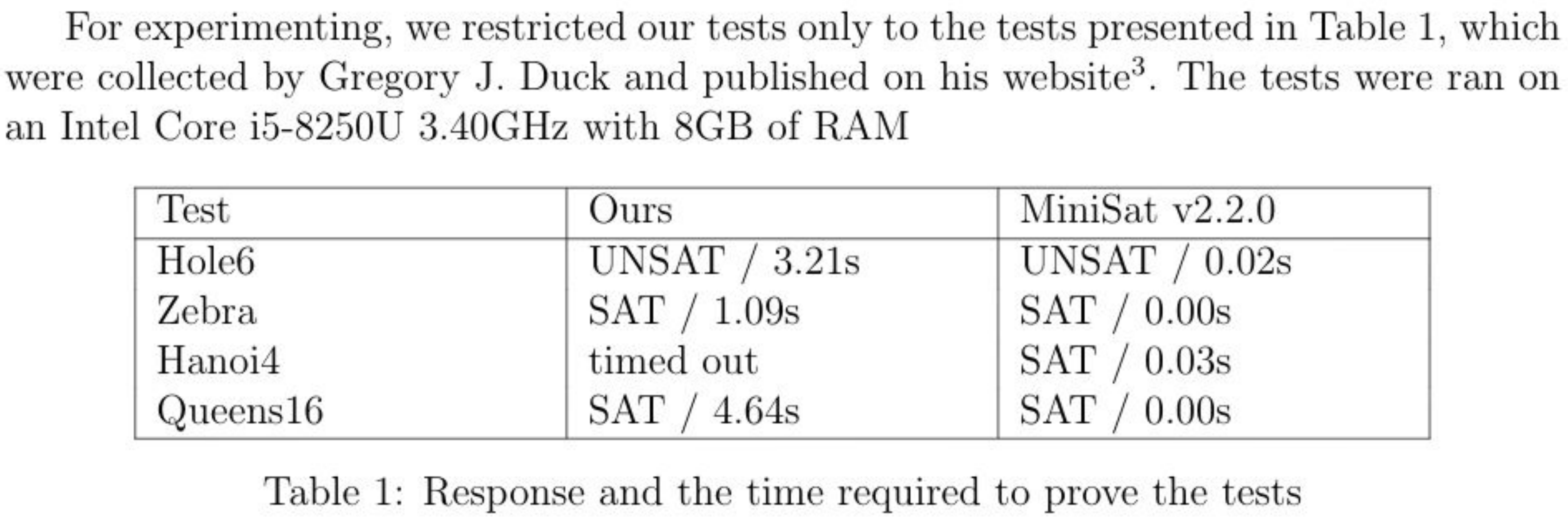
```
lemma forVariableNotSatisfiableExtend_notSatisfiableExtend(
  tau : seq<int>,
  variable : int
)
  requires validVariablesCount();
  requires validClauses();
  requires validValuesTruthAssignment(tau);
  requires validVariable(variable);

  requires ¬isSatisfiableExtend(tau[variable := 0]);
  requires ¬isSatisfiableExtend(tau[variable := 1]);

  ensures ¬isSatisfiableExtend(tau);
{
  if (isSatisfiableExtend(tau)) {
    ghost var tauT := getExtendedCompleteTau(tau);

    if (tauT[variable] = 0) {
      assert isExtendingTau(tau[variable := 0], tauT);
    } else if (tauT[variable] = 1) {
      assert isExtendingTau(tau[variable := 1], tauT);
    }
  }
}
```

Benchmarks



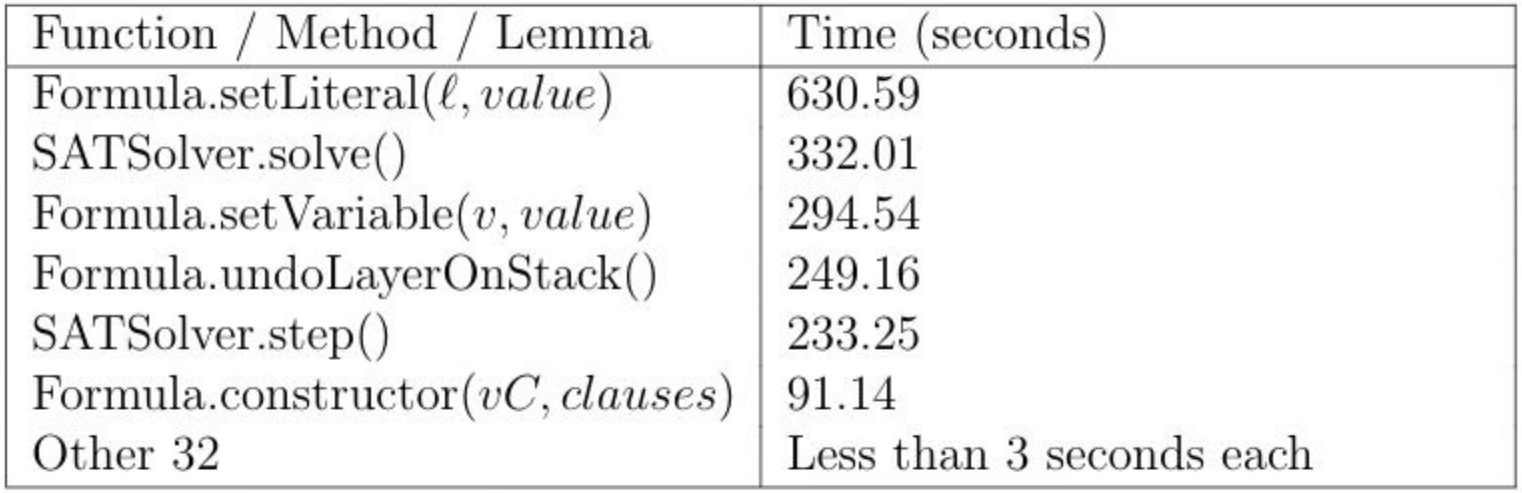
For experimenting, we restricted our tests only to the tests presented in Table 1, which were collected by Gregory J. Duck and published on his website[3]. The tests were ran on an Intel Core i5-8250U 3.40GHz with 8GB of RAM

Test	Ours	MiniSat v2.2.0
Hole6	UNSAT / 3.21s	UNSAT / 0.02s
Zebra	SAT / 1.09s	SAT / 0.00s
Hanoi4	timed out	SAT / 0.03s
Queens16	SAT / 4.64s	SAT / 0.00s

Table 1: Response and the time required to prove the tests

Difficulties working in Dafny

1. The huge proving time.
2. The proof/code ratio is huge. (In undoLayerOnStack for 27 lines of code, there were needed 280 lines of proofs)
3. Imperative code needs a lot of extra proofs.



Function / Method / Lemma	Time (seconds)
Formula.setLiteral(ℓ, *value*)	630.59
SATSolver.solve()	332.01
Formula.setVariable(v, *value*)	294.54
Formula.undoLayerOnStack()	249.16
SATSolver.step()	233.25
Formula.constructor(vC, *clauses*)	91.14
Other 32	Less than 3 seconds each

Table 2: Time required to prove each method / lemma in seconds

Conclusions

We managed to create a formally verified implementation of a SAT Solver, which we think it can be extended to become competitive.

Dafny was a great choice for this project, even if we faced some challenges, it was easy to use and had enough features to permit us to finish the proof of the DPLL algorithm without compromises.

Further work

The current implementation is not competitive yet.

1. Reduce proving time
2. Extend to the CDCL algorithm