LEARNSAT Tutorial

Moti Ben-Ari

http://www.weizmann.ac.il/sci-tea/benari/

Version 2.0

© 2012-17 by Moti Ben-Ari.

This work is licensed under the Creative Commons Attribution-ShareAlike 3.0 License. To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/3.0/; or, (b) send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.

Contents

Chapter 1

Overview

LEARNSAT is a program for learning about SAT solving. It implements the *Davis-Putnam-Logemann-Loveland (DPLL)* algorithm with *conflict-driven clause learning (CDCL)*, *non-chronological backtracking (NCB)* and *lookahead*.

This document is a tutorial on SAT solving with LEARNSAT. It assumes a basic knowledge of propositional logic such as presented in Chapters 2 and 3 of [2]. This is summarized in Chapter 2. For an introduction to SAT solvers, see [2, Chapter 6]. The comprehensive reference is the *Handbook of Satisfiability* [4].

The source programs for LEARNSAT, the examples and the documentation are contained in an archive `learnsat-n.zip`. For instructions on how to run LEARNSAT, see the LEARNSAT *User's Guide and Software Documentation* in the archive.

Chapter 3 demonstrates LEARNSAT in great detail on the example in [8]. Chapter 4 shows SAT solutions for problems in combinatorics, number theory and graph theory. Chapter 5 introduces *bounded model checking*, an important application of SAT solving.

Chapter 2

Propositional logic

In this chapter we give an overview of propositional logic needed to study SAT solving. Sections 2.1 and 2.2 present the syntax and semantics that should be familiar, although clausal form may be new. Section 2.3 gives some important algorithms and theorems concerning clausal form. Section 2.4 introduces the *DPLL* algorithm upon which most SAT solving algorithms are based.

2.1 Syntax

- $\mathscr{P} = \{p_1, p_2, \ldots\}$ is a set of elements p_i called *variables*, *atomic propositions* or *atoms*.
- There is a unary operator *negation*, denoted $\neg$. $\neg x$ is the negation of the atom p. A negated atom is also denoted $\bar{p}$.
- A *literal* is an atom p or the negation of an atom $\neg p$.
- An atom is a *positive literal* and the negation of an atom is a *negative literal*.
- Let l be a literal. l^c, the *complement* of l, is $\neg p$ if $l = p$ for some atom p and is p if $l = \neg p$.
- l and l^c are a *complementary pair of literals*.
- There are two binary operators: *disjunction*, denoted $\vee$, and *conjunction*, denoted $\wedge$.
- A *clause* is the disjunction of a set of literals: $l_1 \vee \cdots \vee l_n$, such as $\neg p_1 \vee \neg p_2 \vee p_3$. A clause can be written using set notation, for example, $\{\bar{p_1}, \bar{p_2}, p_3\}$.
- The *empty clause* $\square$ is the clause that is the empty set.
- (Syntactic definition) A *unit clause* is a clause that has exactly one literal $\{l\}$.

- A *formula* in *conjunctive normal form (CNF)* or in *clausal form* is the conjunction of a set of clauses:

$$(p1 \vee \neg p_2) \wedge (\neg p_1 \vee \neg p_2 \vee p_3) \wedge (\neg p_3).$$

  Nested set notation can be used for formulas in clausal form:

$$\{ \{p1, \bar{p_2}\}, \{\bar{p_1}, \bar{p_2}, p_3\}, \{\bar{p_3}\} \}.$$

- The *empty formula* $\emptyset$ is the formula that is the empty set.

- Let C_1, C_2 be clauses such that $l \in C_1, l^c \in C_2$. C, the *resolvent* of C_1 and C_2, is the clause $(C_1 - \{l\}) \cup (C_2 - \{l^c\})$. C_1 and C_2 are the *parent clauses* of C. C_1 and C_2 are *clashing clauses* that *clash* on the literals l, l^c.

2.2 Semantics

The semantics of a formula in propositional logic are specified by interpretations that assign *truth values* to the atoms, and then the assignment is extended to formulas. The extension is defined in terms of clauses, but they are easy to understand if you translate clauses into disjunctions of literals and formulas into conjunctions of clauses.

- An *interpretation* is a function $\mathscr{I} : \mathscr{P} \to \{T, F\}$.

- The mapping of an interpretation is extended to literals, clauses and formulas as follows:

  - $\mathscr{I}(l) = T$, the literal l is *true* in the interpretation $\mathscr{I}$, if l is an atom p and $\mathscr{I}(p) = T$, or if l is the negation of an atom $\bar{p}$ and $\mathscr{I}(p) = F$. $\mathscr{I}(l) = F$, l is *false* in $\mathscr{I}$, if l is an atom p and $\mathscr{I}(p) = F$, or if l is the negation of an atom $\bar{p}$ and $\mathscr{I}(p) = T$
  - $\mathscr{I}(C) = T$, the clause C is *true* in the interpretation $\mathscr{I}$, if for *at least one* $l \in C$, $\mathscr{I}(l) = T$. $\mathscr{I}(C) = F$, C is *false* in $\mathscr{I}$, if for *all* $l \in C$, $\mathscr{I}(l) = F$.
  - $\mathscr{I}(A) = T$, the formula A is *true* in the interpretation $\mathscr{I}$, if for *all* $C \in A$, $\mathscr{I}(C) = T$. $\mathscr{I}(A) = F$, A is *false* in $\mathscr{I}$, if for *at least one* $C \in A$, $\mathscr{I}(C) = F$.

- A formula A is *satisfiable* if there exists an interpretation $\mathscr{I}$ such that $\mathscr{I}(A) = T$. If there is no such interpretation, A is *unsatisfiable*.

- $\mathscr{I}$ is a *partial interpretation* for A if it maps *some* of the atoms in A into $\{T, F\}$. A partial interpretation can assign truth values to enough atoms so that the truth value of A can be computed. For example, given the formula:

$$A = \{ \{p1, \bar{p_2}\}, \{\bar{p_1}, \bar{p_2}, p_3\}, \{\bar{p_3}\} \},$$

  under the partial interpretation $\{\mathscr{I}(p_2) = F, \mathscr{I}(p_3) = F\}$, $\mathscr{I}(A) = T$, and under the partial interpretation $\{\mathscr{I}(p_3) = T\}$, $\mathscr{I}(A) = F$.

- Let A be a formula and let $\mathscr{I}$ be a partial interpretation for A. $A' = \mathscr{I}(A)$, the formula obtained by *evaluating A under* $\mathscr{I}$, is obtained by removing all clauses $C \in A$ such that for some $l \in C$, $\mathscr{I}(l) = T$, and by removing from the remaining clauses all literals l' such $\mathscr{I}(l') = F$. The justifications for this definition and the next one are given in the Section 2.3.

- (Semantic definition) A clause C is a *unit clause* under a partial interpretation $\mathscr{I}$ if for some $l \in C$, the atom of l is not in the domain of $\mathscr{I}$ and if for all $l' \in C, l' \neq l$, $\mathscr{I}(l') = F$.

- A *decision procedure* for satisfiability is an algorithm that returns *satisfiable* or *unsatisfiable* when a formula is the input to the procedure. When a formula is satisfiable, the decision procedure should return a (possibly partial) interpretation that satisfies the formula.

- A decision procedure is *sound* if the output *satisfiable* is returned only for satisfiable formulas, and it is *complete* if the output *satisfiable* is returned for every satisfiable formula.

- The *SAT* problem is to construct a sound and (hopefully) complete decision procedure for satisfiability. The procedure is called a *SAT solver*. Truth tables can be used as a sound and complete SAT solver, but are always inefficient.

- The SAT problem is $\mathscr{NP}$-complete so it is unlikely that there is an efficient algorithm for satisfiability. Nevertheless, there are algorithms that are efficient on many classes formulas.

2.3 Theorems

Unit clause rule Let A be a formula, $C = \{l\}$ a unit clause in A, and $\mathscr{I}$ a partial interpretation that does not assign a value to l. Construct $\mathscr{I}'$ as the (possibly partial) interpretation obtained by adding an assignment to l to $\mathscr{I}$ such that $\mathscr{I}'(l) = T$. Construct A' by deleting the unit clause C and deleting l^c from every clause in which it appears.

Theorem Let A' be obtained by the unit clause rule from a formula A. A is satisfiable if and only if A' is satisfiable.

Proof Let A be satisfiable under an interpretation $\mathscr{I}_1$ and $C = \{l\} \in A$ a unit clause. Since A is satisfiable, all clauses must evaluate to T, in particular, $\mathscr{I}_1(C) = T$. But $C = \{l\}$ is a unit clause so $\mathscr{I}_1(l) = T$ and also $\mathscr{I}_1(l^c) = F$.

Define $\mathscr{I}_1'$ as the interpretation that is the same as $\mathscr{I}_1$ except that it does not assign a value to the variable of l. We show that A' evaluates to T under this interpretation.

Let $C' \in A'$. We must show that $\mathscr{I}_1'(C') = T$. There are two cases:

- $C' \in A$. C' is not the unit clause which was deleted, so $\mathscr{I}_1$ and $\mathscr{I}_1'$ coincide on the literals of C', and $\mathscr{I}_1'(C') = \mathscr{I}_1(C) = T$.

- $C' = C - \{l^c\}$ for $C \in A$. We showed that $\mathscr{I}_1(l^c) = F$, so $\mathscr{I}_1(C) = T$ only if $\mathscr{I}_1(l') = T$ for some literal $l' \in C, l' \neq l$. But $l' \in C'$ which implies $\mathscr{I}_1'(l') = \mathscr{I}_1(l) = T$.

The proof of the converse is left to the reader.

Resolution rule Let C_1, C_2 be clauses such that $l \in C_1, l^c \in C_2$. C, the *resolvent* of C_1 and C_2, is the clause $(C_1 - \{l\}) \cup (C_2 - \{l^c\})$. C_1 and C_2 are the *parent clauses* of C.

Theorem The resolvent C is satisfiable if and only if the parent clauses C_1, C_2 are both satisfiable.

Proof If C is satisfiable, $\mathscr{I}(l') = T$, for some $l' \in C$, where $l' \neq l$ and $l' \neq l^c$ because l, l^c were deleted by the resolution rule. The literal l' must be an element of C_1 or C_2 (or both); without loss of generality, assume that $l' \in C_1$ so that $\mathscr{I}(C_1) = T$. Extend $\mathscr{I}$ to $\mathscr{I}'$ so that $\mathscr{I}'(l^c) = T$. Then $\mathscr{I}'(C_2) = T$.

If C_1 and C_2 are both satisfiable, at most one of l, l^c can be assigned T; without loss of generality, assume that $\mathscr{I}(l) = T$. Then, for some literal $l' \in C_2, l' \neq l^c$, $\mathscr{I}(l') = T$. But $l' \in C$ so $\mathscr{I}(C) = T$.

Theorem The empty clause is unsatisfiable.

Proof Consider the unsatisfiable formula $p \wedge \neg p$, which is $\{\{p\}, \{\neg p\}\}$ in clausal form. Apply the unit clause rule to $\{p\}$: delete the clause $\{p\}$ and delete the literal $\neg p$ from the clause $\{\neg p\}$. The result is the formula with the single empty clause $\{\{\}\}$, so the empty clause is unsatisfiable.

Theorem The empty formula is satisfiable.

Proof The formula consisting of one atom p, in clausal form $\{\{p\}\}$, is satisfiable. Apply the unit clause rule to obtain the empty formula $\{\}$, which is therefore satisfiable. (Actually, the empty formula is valid but we did not define validity.)

2.4 Davis-Putnam-Logemann-Loveland (DPLL) algorithm

The DPLL algorithm uses the following sub-algorithm:

Unit clause propagation (UCP)
Repeat the unit clause rule generating $A', A'', A''', \ldots$, and $\mathscr{I}', \mathscr{I}'', \mathscr{I}''', \ldots$, until the unit clause rule no longer applies.

DPLL algorithm, specification

Input: A formula A and an empty set. (The empty set is the partial assignment constructed so far.)
Output: Return *unsatisfiable* or a (possibly partial) interpretation that satisfies A.

DPLL algorithm, implementation

$DPLL(A, \mathscr{I})$

- Perform UCP on A under $\mathscr{I}$, resulting in formula A' and partial interpretation $\mathscr{I}'$.
  - If A' contains the empty clause, return *unsatisfiable*;
  - If A' is the empty formula, return $\mathscr{I}'$;
  - (otherwise, continue).
- *Choose* an atom p in A' and a truth value v for p, and let $\mathscr{I}_1$ be the partial interpretation obtained by adding the assignment $p \to v$ to $\mathscr{I}'$, the partial interpretation returned by UCP.
- $result \leftarrow DPLL(A', \mathscr{I}_1)$.
  - If *result* is not *unsatisfiable*, return *result*;
  - (otherwise, continue).
- Let $\mathscr{I}_2$ be the partial interpretation obtained by adding the assignment $p \to v^c$ to $\mathscr{I}'$.
- $result \leftarrow DPLL(A', \mathscr{I}_2)$.
  - Return *result*.

Theorem The DPLL algorithm is sound and complete.

Proof outline We have shown that applying the unit clause rule preserves satisfiability. Aside from assignments made during UCP, the algorithm performs an exhaustive search of assignments to the formula and such a search is sound and complete.

Chapter 3

A detailed example

File: examples.pro

3.1 The DPLL algorithm

For the tutorial, we use the set of clauses from [8] as our example.[1] `examples.pro` contains the predicate `mlm` that runs `dpll` on this set of clauses:

```
mlm :-
  dpll( [  [x1, x031, ~x2], [x1, ~x3], [x2, x3, x4],
           [~x4, ~x5], [x021, ~x4, ~x6], [x5, x6]     ], _).
```

Run the query to display the initial set of clauses:[2]

```
?- mlm.
LearnSAT (version 2.0)
Current set of clauses:
[[x021,~x4,~x6],[x1,x031,~x2],[x1,~x3],[x2,x3,x4],[x5,x6],[~x4,~x5]]
```

There are no units, so a decision assignment must be made:

```
Decision assignment: x021=0
Current set of clauses:
[[x1,x031,~x2],[x1,~x3],[x2,x3,x4],[x5,x6],[~x4,~x5],[~x4,~x6]]
```

The clause `[~x4,~x6]` results because the literal `x021` received the value `false` and was deleted from the first clause.

[1] LEARNSAT makes decision assignments in lexicographic order, so `x21` and `x31` have a `0` added to their names to preserve the order in [8]. You can specify a different order using the `set_order` command.

[2] The non-default display option `clause` must be set.

Next, decision assignments are made to x031 and x1:

```
Decision assignment: x031=0
Current set of clauses:
[[x1,~x2],[x1,~x3],[x2,x3,x4],[x5,x6],[~x4,~x5],[~x4,~x6]]
Decision assignment: x1=0
Current set of clauses:
[[x2,x3,x4],[x5,x6],[~x2],[~x3],[~x4,~x5],[~x4,~x6]]
```

Finally we have a unit clauses and unit propagation can be performed:

```
Propagate unit: ~x2 (x2=0) derived from: [x1,x031,~x2]
Current set of clauses:
[[x3,x4],[x5,x6],[~x3],[~x4,~x5],[~x4,~x6]]
Propagate unit: ~x3 (x3=0) derived from: [x1,~x3]
Current set of clauses:
[[x4],[x5,x6],[~x4,~x5],[~x4,~x6]]
```

Unit propagation of [~x2] and [~x3] results in the creation of another unit clause [x4] so unit propagation continues:

```
Propagate unit:  x4 (x4=1) derived from: [x2,x3,x4]
Current set of clauses:
[[x5,x6],[~x5],[~x6]]
Propagate unit: ~x5 (x5=0) derived from: [~x4,~x5]
Current set of clauses:
[[x6],[~x6]]
Propagate unit: ~x6 (x6=0) derived from: [x021,~x4,~x6]
Conflict clause: [x5,x6]
```

The unit clauses [~x5] and [~x6] falsify the clause [x5,x6], which is called a *conflict clause*: it shows that the current assignment does not satisfy the initial set of clauses. Frequently, evaluating a formula under a partial assignment will result in a conflict clause, so the partial assignment cannot be part of any satisfying assignment.

Now, the algorithm backtracks, trying the assignment of 1 to x1 and then 0 to x2 and x3. Again, a conflict clause is encountered:

```
Decision assignment: x1=1
Current set of clauses:
[[x2,x3,x4],[x5,x6],[~x4,~x5],[~x4,~x6]]
Decision assignment: x2=0
Current set of clauses:
```

```
[[x3,x4],[x5,x6],[~x4,~x5],[~x4,~x6]]
Decision assignment: x3=0
Current set of clauses:
[[x4],[x5,x6],[~x4,~x5],[~x4,~x6]]
Propagate unit:  x4 (x4=1) derived from: [x2,x3,x4]
Current set of clauses:
[[x5,x6],[~x5],[~x6]]
Propagate unit: ~x5 (x5=0) derived from: [~x4,~x5]
Current set of clauses:
[[x6],[~x6]]
Propagate unit: ~x6 (x6=0) derived from: [x021,~x4,~x6]
Conflict clause: [x5,x6]
```

The algorithm backtracks again, assigning 1 to x3, followed by 0 to x4 and x5. Then, unit propagation results in assigning 1 to x6. No conflict clauses are encountered and all variables have been assigned to, so the algorithm terminates and displays the satisfying assignment:

```
Decision assignment: x3=1
Variables: [x3,x4,x5,x6]
Current set of clauses:
[[x5,x6],[~x4,~x5],[~x4,~x6]]
Decision assignment: x4=0
Variables: [x4,x5,x6]
Current set of clauses:
[[x5,x6]]
Decision assignment: x5=0
Variables: [x5,x6]
Current set of clauses:
[[x6]]
Propagate unit:  x6 (x6=1) derived from: [x5,x6]
Satisfying assignments:
[x021=0,x031=0,x1=1,x2=0,x3=1,x4=0,x5=0,x6=1]
Statistics: clauses=6, variables=8, units=9, decisions=9, conflicts=2
```

3.2 Display options

The LEARNSAT User's Guide gives the two dozen display options that you can set and clear. Here we mention a few of them that can be useful when initially studying SAT solving.

Display option `clause` shows the current set of clauses after evaluation under the partial assignment as shown above.

The satisfying assignments are display as a sorted list. If the (default) `sorted` option is cleared, the assignments will be displayed in the reverse order in which they were made:

```
Satisfying assignments:
[x6=1,x5=0,x4=0,x3=1,x2=0,x1=1,x031=0,x021=0]
```

Display option `variable` shows the set of *unassigned* variables before each decision assignment and `partial` shows the set of assignments that have been made *so far*:

```
Variables: [x021,x031,x1,x2,x3,x4,x5,x6]
Decision assignment: x021=0
Assignments so far:
[x021=0]
Variables: [x031,x1,x2,x3,x4,x5,x6]
Decision assignment: x031=0
Assignments so far:
[x031=0,x021=0]
Variables: [x1,x2,x3,x4,x5,x6]
Decision assignment: x1=0
Assignments so far:
[x1=0,x031=0,x021=0]
Propagate unit: ~x2 (x2=0) derived from: [x1,x031,~x2]
Assignments so far:
[x2=0,x1=0,x031=0,x021=0]
```

Display option `assignment` shows the assignments that caused a conflict clause:

```
Conflict clause: [x5,x6]
Conflict caused by assignments:
[x021=0,x031=0,x1=0,x2=0,x3=0,x4=1,x5=0,x6=0]
```

3.3 Lookahead

Once unit clause propagation has been performed, a variable must be chosen and assigned a value. Lookahead algorithms use heuristics to choose the next variable to be assigned. The lookahead default `none` causes LEARNSAT to choose variables in alphanumeric order. You can also specify the order using `set_order`. LEARNSAT implements two simple heuristics:

- `current`: Examine the clauses remaining after evaluation under the current partial assignment and choose a variable that occurs most often.

- `original`: Compute the number of occurrences of each variable in the original set of clauses. Choose a variable with the most occurrences among the unassigned variables.

The rationale is that extended a partial assignment by assigning a value to a variable that occurs frequently will likely cause the formula to become satisfied, create a conflict clause or generate many unit clauses.

For the example in Section 3.1, the variable appearing most often is `x4`. Assigning first to this variable results in a satisfying assignment without encountering conflict clauses and backtracking! Display option `look` shows a list of the number of occurrences of each variable in descending order; each element is of the form `N-Var`:

```
LearnSAT (version 2.0)
Current set of clauses:
[[x021,~x4,~x6],[x1,x031,~x2],[x1,~x3],[x2,x3,x4],[x5,x6],[~x4,~x5]]
Occurrences of variables: [3-x4,2-x1,2-x2,2-x3,2-x5,2-x6,1-x021,1-x031]
Decision assignment: x4=0
Current set of clauses:
[[x1,x031,~x2],[x1,~x3],[x2,x3],[x5,x6]]
Occurrences of variables: [2-x1,2-x2,2-x3,1-x031,1-x5,1-x6]
Decision assignment: x1=0
Current set of clauses:
[[x031,~x2],[x2,x3],[x5,x6],[~x3]]
Propagate unit: ~x3 (x3=0) derived from: [x1,~x3]
Current set of clauses:
[[x031,~x2],[x2],[x5,x6]]
Propagate unit:  x2 (x2=1) derived from: [x2,x3,x4]
Current set of clauses:
[[x031],[x5,x6]]
Propagate unit:  x031 (x031=1) derived from: [x1,x031,~x2]
Current set of clauses:
[[x5,x6]]
Occurrences of variables: [1-x5,1-x6]
Decision assignment: x5=0
Current set of clauses:
[[x6]]
Propagate unit:  x6 (x6=1) derived from: [x5,x6]
Satisfying assignments:
[x031=1,x1=0,x2=1,x3=0,x4=0,x5=0,x6=1]
Statistics: clauses=6, variables=8, units=4, decisions=3, conflicts=0
```

The choice of the lookahead mode is independent of the choice of the algorithmic mode (see below). Here are the statistics of this example with and without `current` lookahead:

```
dpll+none:    units=9,  decisions=9, conflicts=2
dpll+current: units=3,  decisions=4, conflicts=0

cdcl+none:    units=8,  decisions=6, conflicts=1
cdcl+current: units=3,  decisions=4, conflicts=0

ncb+none:     units=10, decisions=6, conflicts=1
ncb+current:  units=3,  decisions=4, conflicts=0
```

For this formula, lookahead always results in better performance, but this does not hold for all the examples in the archive.

You are invited to implement other lookahead algorithms:

- LEARNSAT always assigns first false and then true to the chosen variable. Choose the *literal* that occurs most often and assign the value that makes the literal true.
- Choose the variable or literal that occurs most often in the *shortest clauses*. This makes it more likely unit clauses will be generated.

Lookahead algorithms are described in [6]; see [5] for an interesting application.

3.4 Assignment trees

Display option `tree` generates a tree of the assignments (Figure 3.1). Display option `label` adds the antecedent clauses to nodes that implied by unit propagation. Decision assignments are red; assignments implied by unit propagation are black; conflict nodes have a red double border; the nodes where the satisfying assignments are found have a green double border.[3] The search backtracked to assign 1 to `x1` after trying 0 and to assign 1 to `x3` after trying 0.

Display option `tree_inc` generates the trees incrementally whenever a conflict is reached.

[3]The graphs are decorated in color but you can request that LEARNSAT decorate graphs in black and white (`set_decorate_mode(bw)`): decision nodes are bold and satisfying assignments have a triple border.

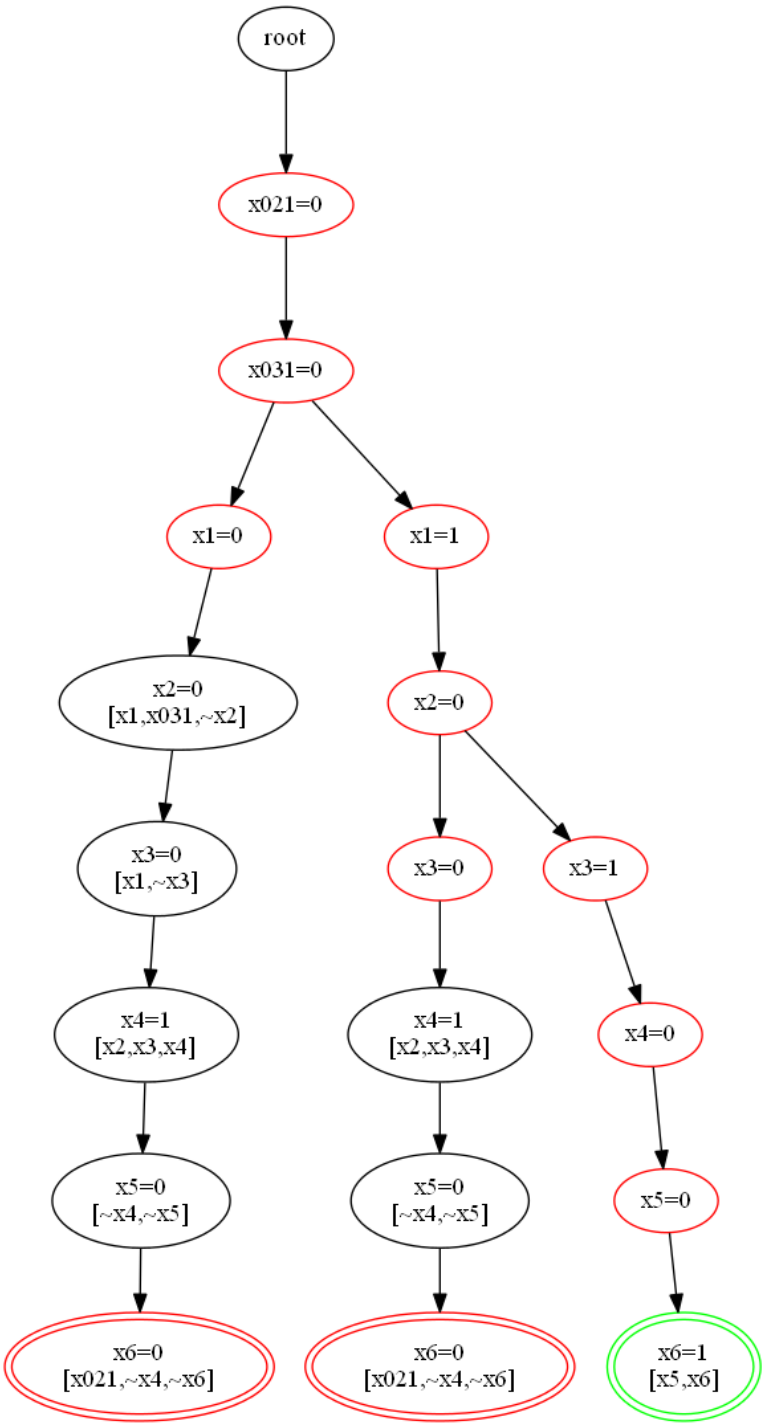


Figure 3.1: Tree of assignments for the DPLL algorithm

3.5 Conflict-directed clause learning

Select CDCL mode:

```
?- set_mode(cdcl).
```

The display options relevant to this mode selected by default are: `learned`, `resolvent`, `uip`.

After encountering the conflict clause `[x5,x6]`, the algorithm *learns* the clause `[x021,~x4]`, as described in Sections 3.7–3.8. When backtracking to assign 1 to `x3`, the learned clause becomes a unit clause based upon the previous assignment of 0 to `x021`. Unit propagation leads immediately to a satisfying assignment:

```
Conflict clause: [x5,x6]
  . . .
Learned clause from resolution (used): [x021,~x4]
Decision assignment: x1=1@3
Propagate unit: ~x4 (x4=0@3) derived from: [x021,~x4]
Decision assignment: x2=0@4
Propagate unit:  x3 (x3=1@4) derived from: [x2,x3,x4]
Decision assignment: x5=0@5
Propagate unit:  x6 (x6=1@5) derived from: [x5,x6]
Satisfying assignments:
[x021=0@1,x031=0@2,x1=1@3,x2=0@4,x3=1@4,x4=0@3,x5=0@5,x6=1@5]
Statistics: clauses=6, variables=8,
  units=8, decisions=6, conflicts=1, learned clauses=1
```

Assignments are displayed together with their levels. Each decision assignment increases the level, while assignments implied by unit propagation receive the level of the last decision assignment.

The search with CDCL is more efficient: compare the statistics (6 decisions and 1 conflict instead of 9 decisions and 2 conflicts) or the the assignment trees (Figures 3.1–3.2).

For lookahead, occurrences of variables in the learned clauses are taken into account.

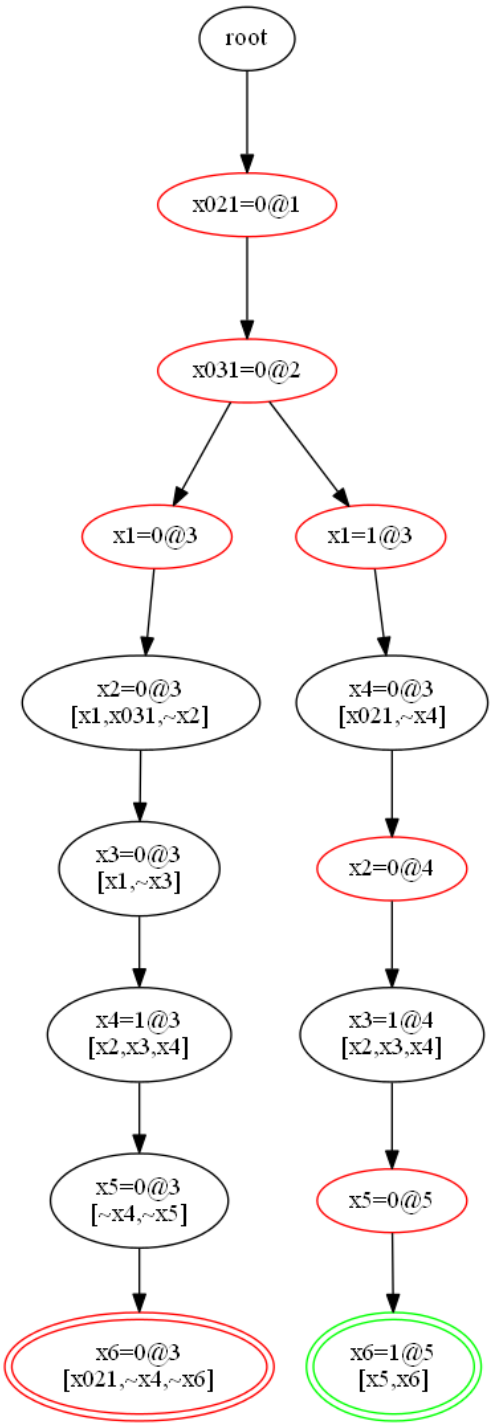


Figure 3.2: Tree of assignments for the DPLL algorithm with CDCL

3.6 The implication graph

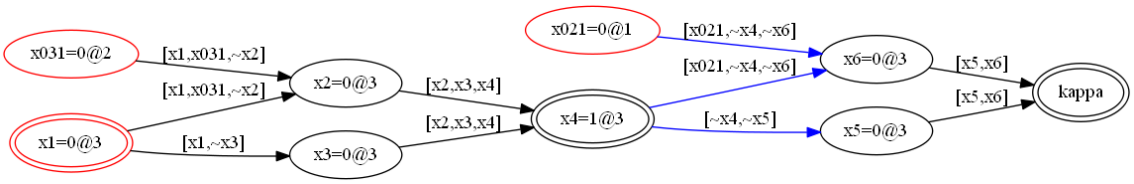
When a conflict clause is found, an *implication graph* is constructed and used to learn a clause.

Display options for generating a implication graph

Display option `graph` displays a textual representation of the implication graph: a list of nodes and a list of edges.

```
Implication graph:
[kappa, x021=0@1, x031=0@2, x1=0@3, x2=0@3, x3=0@3, x4=1@3, x5=0@3, x6=0@3]
[
x021=0@1 --[x021,~x4,~x6]--> x6=0@3,
x031=0@2 --[x1,x031,~x2]--> x2=0@3,
x1=0@3 --[x1,x031,~x2]--> x2=0@3,
x1=0@3 --[x1,~x3]--> x3=0@3,
x2=0@3 --[x2,x3,x4]--> x4=1@3,
x3=0@3 --[x2,x3,x4]--> x4=1@3,
x4=1@3 --[~x4,~x5]--> x5=0@3,
x4=1@3 --[x021,~x4,~x6]--> x6=0@3,
x5=0@3 --[x5,x6]--> kappa,
x6=0@3 --[x5,x6]--> kappa
]
```

Display option `dot` generates a DOT representation of the graph and display option `label` causes the edges to be labeled with the antecedent clauses:



Display options `incremental` and `dot_inc` generate the graphs after each step of the algorithm, not just when a conflict clause is reached.

The meaning of an implication graph

The implication graph represents the result of unit propagation that leads to a conflict, starting from a set of decision assignments. In the graph, are three source nodes, one for each of the decision assignments (`x021=0@1`, `x031=0@2`, `x1=0@3`) that together lead to a conflict by unit propagation.

The conflict is represented by the sink node `kappa` and the assignments implied by unit propagation are represented by internal nodes labeled with the assignments. An edge of the graph is labeled with the *antecedent* clause of its target node. This is the unit clause that implied the assignment labeling that node. For example, the decision assignments `x1=0@3`, `x031=0@2` cause `[x1,x031,~x2]` to become a unit clause `[~x2]` and therefore this clause implies the assignment of 0 to `x2`. The assignment receives the same level as the last decision assignment so its node is labeled `x2=0@3`.

For each non-source node, there is an incoming edge for each literal in the unit clause except the one whose assignment is implied. The source nodes of these edges are the ones labeled with the assignments to those literals. For example, the clause `[x1,x031,~x2]` with three literals implied the assignment `x2=0@3`; therefore, the node labeled `x2=0@3` has two incoming edges: one from the node `x1=0@3` that assigned 0 to `x1` and one from the node `x031@2` that assigned 0 to `x031`.

The implication graph shows that the decision assignments `x021=0@1`, `x031=0@2`, `x1=0@3` cause a conflict. Clearly, if `[x021,x031,x1]` were a clause in the original set of clauses, it would be evaluated when the decisions were made. Since it evaluates to 0, it is a conflict clause that would be found immediately and there would be no need to carry out unit propagation. Therefore, by *learning* this clause—adding it to our original set of clauses—any subsequent decision assignments that include these three would *immediately* lead to the discovery of a conflict.

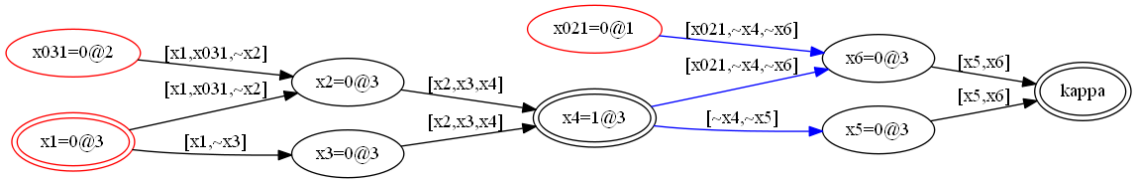
However, there is no advantage to learning a clause based only on decision assignments, because the same set of decisions at levels 1, 2 and 3 will never occur again after backtracking.

Sections 3.7–3.8 present algorithms that can find *shorter* learned clauses that are more likely to become conflict clauses as the algorithm continues. Section 3.7 describes how to learn a clause by computing a dominator in the implication graph and Section 3.8 describes how to learn a clause resolving backwards from the conflict node.

The learned clause used by LEARNSAT is the one that is learned by resolution. The display options `learned`, `resolvent` and `uip` trace the computation by resolution. Display option `dominator` enables the computation and display of a dominator.

3.7 Learning a clause from a dominator

Here again is the implication graph:



Consider all the (four) paths from the *last* decision assignment, `x1=0@3`, the decision assignment at the highest level, to the node labeled `kappa`. The node `x4=1@3` is called a *dominator* of a decision node if it appears on all the paths from the decision node to `kappa`. LEARNSAT computes the paths and finds a dominator:

```
Paths from the decision node at this level to kappa:
x1=0@3 --> x2=0@3 --> x4=1@3 --> x5=0@3 --> kappa
x1=0@3 --> x2=0@3 --> x4=1@3 --> x6=0@3 --> kappa
x1=0@3 --> x3=0@3 --> x4=1@3 --> x5=0@3 --> kappa
x1=0@3 --> x3=0@3 --> x4=1@3 --> x6=0@3 --> kappa
A dominator is: x4=1@3
```

A dominator is a *unique implication point (UIP)* because its assignment participates in the conflict in the same way as do *all* the decision assignments that it dominates. Here, the assignment `x4=1@3` is implied by the *two* assignments `x031=0@2` and `x1=0@3`, so `~x4` can replace `x031` and `x1` in the learned clause `[x021,x031,x1]` to obtain a shorter learned clause `[x021,~x4]`.

There remains one path from a decision node to `kappa` that does not contain the dominator:

```
Paths from decision nodes at lower levels to kappa:
x021=0@1 --> x6=0@3 --> kappa
```

The learned clause is therefore composed of the complement of the assignment at the UIP, together with the complements of assignments at *higher* levels that are not dominated by the UIP. The literal chosen for this path is the one just before the path joins an assignment at the lowest level; in this case, it is the decision assignment to `x021`:

```
Learned clause from dominator (not used): [x021,~x4]
```

The blue lines form a *cut*: their source nodes are the assignments that define the literals in the learned clause. These nodes define a learned clause because assignments to the literals necessarily lead to the conflict. The cut we chose includes the edges whose source node is the UIP together with an edge on each path from a decision node that is not dominated by the UIP.

3.8 Learning a clause by resolution

The learned clause can be obtained by resolution starting with the conflict clause—the antecedent clause of the `kappa` node—and terminating when a UIP is found. The conflict clause as the initial *current clause*. At each step, the antecedent clause at clashes with the current clause on the literal that was assigned at that node by unit propagation. For example, given the initial current clause `[x5,x6]`, the conflict resulted from the assignment of 0 to `x5` which was forced by the assignment of 1 to `~x5`, the literal in the unit clause `[~x4,~x5]`. The two clauses have clashing literals, so we can resolve `[x5,x6]` with the `[~x4,~x5]` to obtain the resolvent `[~x4,x6]` which becomes the new current clause.

The next step is to resolve this clause with `[x021,~x4,~x6]`, the antecedent of the assignment to `x6`, to obtain `[x021,~x4]`. The resolution now terminates because this clause is associated with a UIP: there is one (implied) literal assigned at the current level and the other literals were assigned by decision assignments at higher levels.

Here is the computation as displayed by LEARNSAT:

```
Conflict clause: [x5,x6]
Not a UIP: two literals [x6,x5] are assigned at level: 3
Resolvent: of [x5,x6] and antecedent [x021,~x4,~x6] is [x5,x021,~x4]
Not a UIP: two literals [~x4,x5] are assigned at level: 3
Resolvent: of [x5,x021,~x4] and antecedent [~x4,~x5] is [x021,~x4]
UIP: one literal ~x4 is assigned at level: 3
Learned clause from resolution (used): [x021,~x4]
```

Since the conflict clause is unsatisfiable under the current set of decision assignments, so is the set consisting of it and its clashing clause, and therefore their resolvent—the next current clause—is also unsatisfiable.

For example, `[x5,x6]` is a conflict clause because it evaluates to 0. The antecedent clause `[~x4,~x5]` is a unit clause and therefore exactly one literal is assigned 1. Since both literals in the conflict clause `[x5,x6]`, are assigned 0, `~x5`, the complement of `x5`, is assigned 1, so it must be the *only* literal in the antecedent clause `[~x4,~x5]` that is assigned 1. Therefore, `~x4` is assigned 0 and we conclude that all literals in the resolvent `[~x4,x6]` are assigned 0.

Which of these clauses should we learn? Successive resolution steps will eventually lead back to the clause defined by the decision nodes (`[x021,x031,x1]`), but there is no advantage to learning this clause. However, the resolvent clause associated with a UIP (`[x021,~x4]`) contains only a single literal at the current level (`~x4`), so it can replace the decision literal at the same level (`x1`), as well as any decision variables at higher levels that are dominated by the UIP (`x031`).

3.9 Non-chronological backtracking

Select NCB mode:

```
?- set_mode(ncb).
```

When a learned clause has been obtained, the backtrack level for non-chronological backtracking is computed. This is the highest level of an assignment in the learned clause except for the current level. In the example, the learned clause is `[x021,~x4]`, where `x4` was assigned at level 3, the current level, and `x021` was assigned at level 1, which becomes the backtrack level. When backtracking, we can skip decision assignments whose level is at a higher level than the backtrack level, in the example, the decision assignments at levels 3 and 2:

```
Non-chronological backtracking to level: 1
Skip decision assignment: x1=1@3
Skip decision assignment: x031=1@2
```

Once 0 is assigned to `x021` at level 1, the clause `[x021,~x4]` becomes a unit clause `[~x4]` which forces the assignment of 0 to `x4`. Consider now the original set of clauses:

```
[[x1,x031,~x2], [x1,~x3], [x2,x3,x4], [~x4,~x5], [x021,~x4,~x6], [x5,x6]]
```

Under these two assignments, the set becomes:

```
[[x1,x031,~x2], [x1,~x3], [x2,x3], [x5,x6]]
```

The next decision assignment was 0 to `x031` giving:

```
[[x1,~x2], [x1,~x3], [x2,x3], [x5,x6]]
```

If 0 is assigned to `x1`, `[x2,x3]` becomes a conflict clause. It follows that 1 must be assigned to `x1` and there is no need to *flip* the variable (assign the other Boolean value to the variable); therefore, we can skip over `x1` when backtracking.

Once we know that 1 is assigned to `x1`, the clause `[x1,x031,~x2]` becomes true and can be deleted. Since that clause contains the only occurrence of the variable `x031`, it follows that if there exists a satisfying assignment, it is independent of the value assigned to `x031`, so, again, we can skip over `x031` when backtracking.

From the learned clause, it follows that *the assignment of 0 to* `x021` *by itself makes it unnecessary to flip the variables* `x031` *and* `x1`. We can backtrack directly to flip the variable `x021`.

NCB can be seen in the tree of assignments (Figure 3.3). The graph is no longer a tree but a DAG (directed acyclic graph) because once `x021` is assigned (0 or 1), the assignments of 0 to `x031` and then `x1` lead to the same sequence of unit propagations. The graph branches only at its leaves because one assignment leads to a conflict clause and the other to a satisfying assignment.

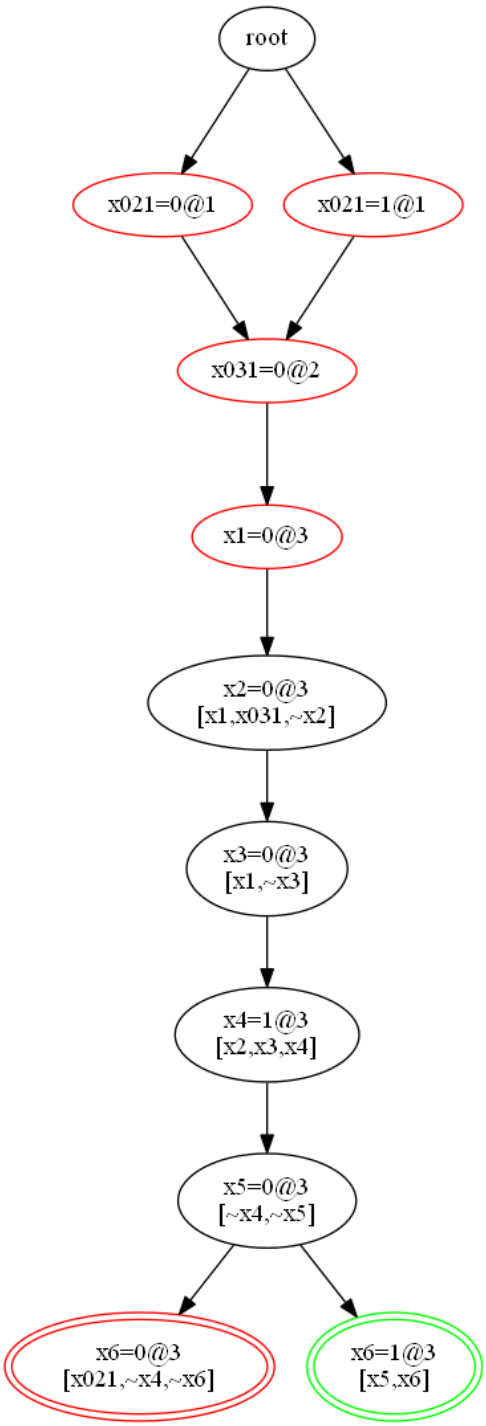


Figure 3.3: Tree of assignments for the DPLL algorithm with CDCL and NCB

Chapter 4

Combinatorics, numbers, graphs

Problems from combinatorics are a rich source of examples for SAT solving. The chapter starts with the well-known pigeonhole principle, which is examined in detail. This is followed by the classic n-queens problem and the solution of Sudoku puzzles. Four problems from number theory follow: Ramsey theory, Schur triples, Langford's problem and van der Waerden's problem. These were inspired by the presentations in [5, 7]. The next two problems are more complex: pebbling problems and seating guests at a table under constraints. SAT solvers can be used on graphs. In Section 4.10, we show how a complex planar graph can be colored using four colors. The existence of such a coloring following from the Four Color Theorem that was proved relative recently. Section 4.11 gives a variant of the n-queens problem: can you placed colored queens on every square such that no row, column and diagonal contains queens of the same color. Section 4.12 presents Tseitin graphs that used studies for resolution.

4.1 The pigeonhole principle

File: `pigeon.pro`

The pigeonhole principle states that if we place $n+1$ pigeons in n holes then one hole contains at least two pigeons. Therefore, a formula stating that all pigeons are in different holes must be unsatisfiable. Let `pij` be an atom whose intended meaning is that pigeon `i` is in hole `j`; for two holes and three pigeons, the set of clauses is:

```
[p11, p12],   [p21, p22],   [p31, p32],   % Each pigeon in hole 1 or 2
[~p11, ~p21], [~p11, ~p31], [~p21, ~p31], % No pair is in hole 1
[~p12, ~p22], [~p12, ~p32], [~p22, ~p32], % No pair is in hole 2
```

For three holes and four pigeons the formula is:

```
% Each pigeon in at least hole
[p11, p12, p13], [p21, p22, p23], [p31, p32, p33], [p41, p42, p43],
% Each hole has at most one pigeon
[~p11, ~p21], [~p11, ~p31], [~p11, ~p41],
[~p21, ~p31], [~p21, ~p41], [~p31, ~p41],
[~p12, ~p22], [~p12, ~p32], [~p12, ~p42],
[~p22, ~p32], [~p22, ~p42], [~p32, ~p42],
[~p13, ~p23], [~p13, ~p33], [~p13, ~p43],
[~p23, ~p33], [~p23, ~p43], [~p33, ~p43]
```

For the two-hole formula, the mode chosen makes no difference, but for the three-hole formula, NCB makes a significant improvement:

```
dpll: units=49, decisions=16, conflicts=9
cdcl: units=42, decisions=16, conflicts=9, learned clauses=6
ncb:  units=25, decisions=10, conflicts=3, learned clauses=3
```

If you look at the trace in NCB mode, the following clauses are learned:

```
[p43,p33,~p22],   [p42,p32,~p23],  [p41,p31,~p23]
```

Six decision assignments are skipped—both 0 and 1 for the variables `p12`, `p13` and `p22`—so there are only three conflict instead of nine.

Let us look now at the implication graphs. The graph for the two-hole formula is shown in Figure 4.1 (shown top-to-bottom instead of left-to-right). The conflict clause is `[~p12,~p32]` (stating that either pigeon 1 or pigeon 3 is not in hole 2) and the dominator is the node labeled `p12=1@1`. Since *all* nodes are assigned at level 1, by resolving backwards we should be able to learn the (unit) clause `[~p12]`.

However, we must be careful. According to [8, p. 137]: "at each step i, a literal l assigned at the current decision level d is selected and the intermediate clause (...) is resolved with the antecedent of l." If we *select* `~p12` and resolve `[~p12,~p32]` with `[p11,p12]`, the antecedent of the second node from the top, the result is `[p11,~p32]`, which is actually worse than just taking the decision node `[p11]`. When selecting the literal to resolve, we must be careful to choose the *last* one that was assigned, here `~p32`. If this is done, the sequence of resolvents is:

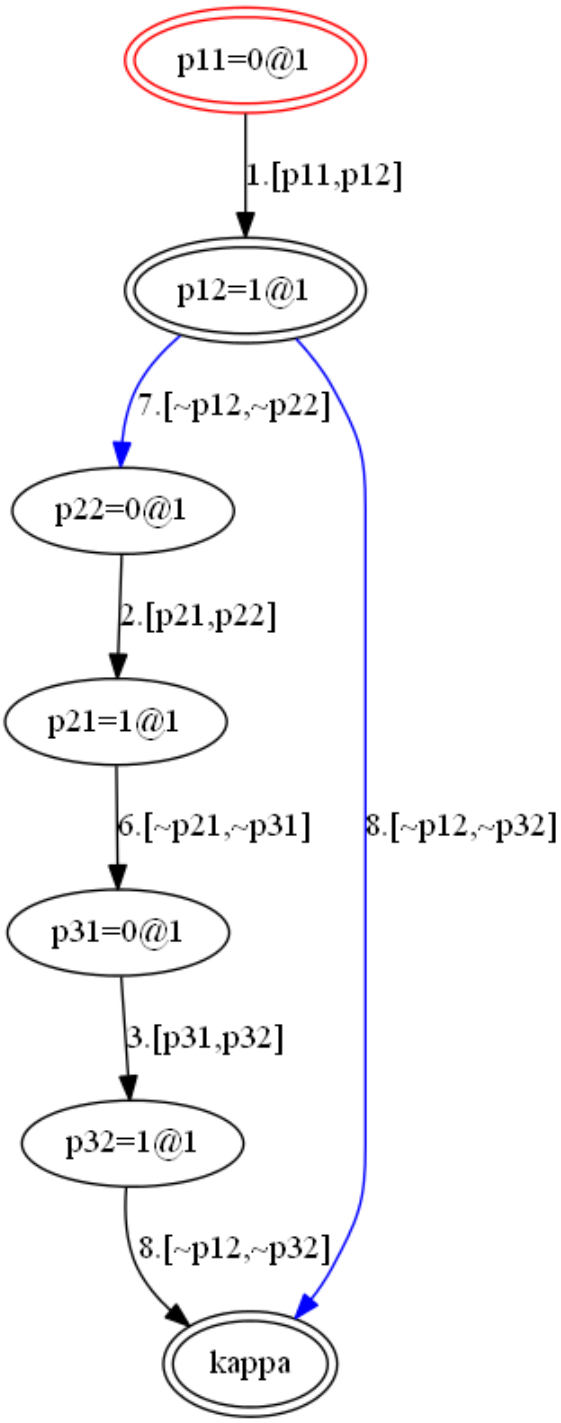


Figure 4.1: Implication graph for two-hole pigeonhole

```
Conflict clause: [~p12,~p32]
Resolvent: of [~p12,~p32] and antecedent [p31,p32] is [~p12,p31]
Resolvent: of [~p12,p31] and antecedent [~p21,~p31] is [~p12,~p21]
Resolvent: of [~p12,~p21] and antecedent [p21,p22] is [~p12,p22]
Resolvent: of [~p12,p22] and antecedent [~p12,~p22] is [~p12]
UIP: one literal ~p12 is assigned at level: 1
Learned clause from resolution: [~p12]
```

The implication graph for the three-hole formula is shown in Figure 4.2. The node labeled `p22=1@3` is a dominator and the clause learned by resolution is `[p43,p33,~p22]`. The blue lines shows the *cut*: it is sufficient to assign to the three literals before the cut in order to obtain a conflict, and the complements of these literals form the learned clause.

The literals that form part of the learned clause come from assignments at higher levels that are *not* decision assignments, but assignments implied by unit propagation: `p43` and `p33` which are assigned at level 2.

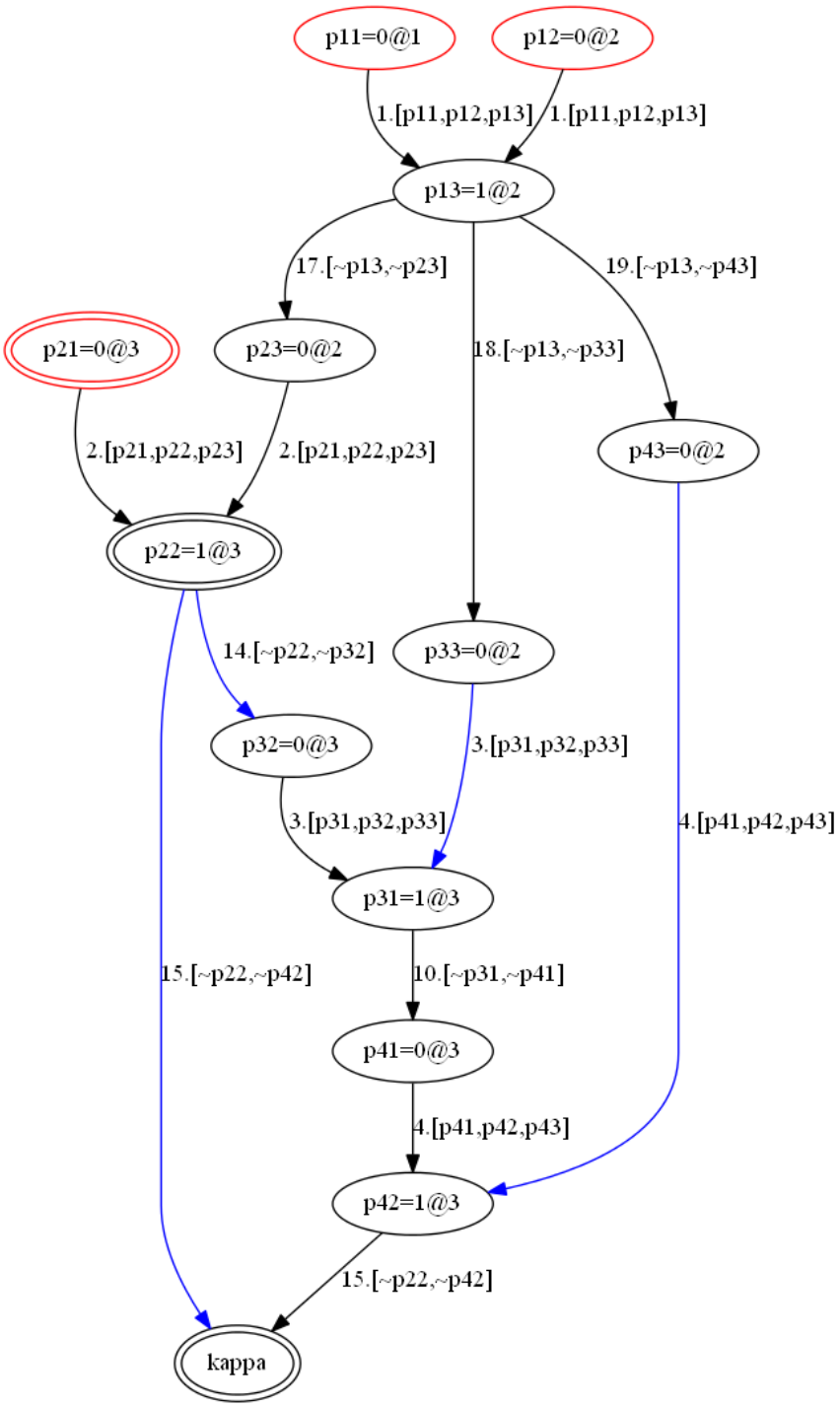
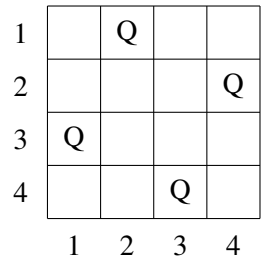


Figure 4.2: Implication graph for three-hole pigeonhole

4.2 The n-queens problem

File: queens.pro

The n-queens problem is to place n queens on an $n \times n$ chess board such that no queen can capture another. The archive includes programs for both 4 queens and 8 queens. Here we look at a 4×4 board. A solution is:



The problem can be encoded in clausal form using 80 clauses on 16 variables [2, Section 6.4]. Other encodings are discussed in [11, Section 2.3.1] based on [10]. The 4-queens problem has a solution (in fact, two solutions), so the formula will be satisfiable and the satisfying assignments provide a solution:

```
[p11=0,p12=0,p13=1,p14=0,p21=1,p22=0,p23=0,p24=0,
 p31=0,p32=0,p33=0,p34=1,p41=0,p42=1,p43=0,p44=0]
```

where `pij` is true if a queen is placed in column `i` and row `j`.

The statistics for the different modes are:

```
dpll: units=30, decisions=6, conflicts=2
cdcd: units=30, decisions=6, conflicts=2, learned clauses=2
ncb:  units=25, decisions=5, conflicts=1, learned clauses=1
```

If you examine the assignment tree for the DPLL algorithm (not shown), you will see the effectiveness of unit propagation. Just three decision assignments (`p11=0@1`, `p12=0@2`, `p13=0@3`) imply the assignment `p14=1@3`, which in turn implies more assignments of 0 along its row and diagonal. The placement of the first queen is shown on the left of Figure 4.3, where the subscripts denote the assignment level and primes denote the inferred assignments.

The next decision assignment is `p21=0@4` or `p21=1@4`, both of which lead to conflicts. Let us look at `p21=0@4`. `p21=0@4` implies `p22=1@4` because there has to be a queen in the second column. The next implications are `p42=0@4`, `p43=1@4`, `p33=0@4`, `p31=1@4` (see the right of Figure 4.3). Clearly, there is a conflict because the clause `[~p22,~p31]` is false. The intended meaning of this clause is that either there is no queen in (column 2, row 2) or there is no queen in (column 3, row 1) because they can capture each other.

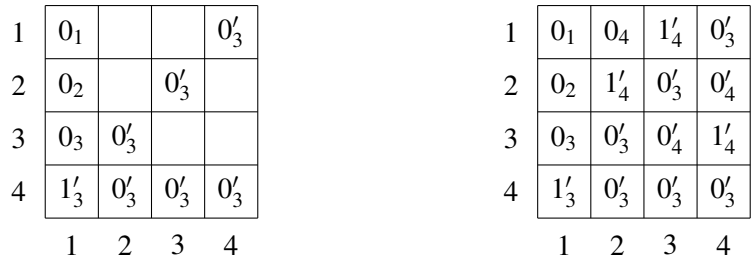


Figure 4.3: Decisions and implications for the four-queens problem

We leave it to the reader to examine the sequence of assignments following the decision assignment `p21=1@4` and to show that it too leads to a conflict.

Figure 4.4 shows the implication graph with the dominator node `p22=1@4` and the cut that defines the learned clause `[~p22,p32,p34,p41,p44]`. The learned clause is such that the decision assignments at the higher levels (`p11=0@1`, `p12=0@2`, `p13=0@3`) make it a unit clause `[~p22]` and force the assignment `p22=1@4`, regardless of the assignment to `p21`. Therefore, non-chronological backtracking can skip the assignment `p21=1@4` and continue with the previous decision assignment. The assignment `p13=1@3` instead of `p13=0@3` leads to a satisfying assignment by unit propagation alone!

4.3 Sudoku

`File: sudoku.pro`

The encoding of an $n \times n$ Sudoku puzzle into CNF is straightforward but requires quite a few clauses. You need clauses to express that each of the n numbers appear in each row, column and block, and that no number appears twice. The archive includes two CNF formulas for a 4×4 Sudoku puzzle: one with the full CNF and one with a simpler CNF. The simpler CNF does not include that clauses specifying that no number appear twice in each row, column and block, because if all n numbers appear once, than none appears twice. You can experiment to determine which encoding is more efficient.

The CNF formulas were generated using a program by Ivor Spence that outputs DIMACS, which was converted to the LEARNSAT CNF format using the program `dimacs.pro`.
See: `http://www.cs.qub.ac.uk/~I.Spence/SuDoku/SuDoku.html`.

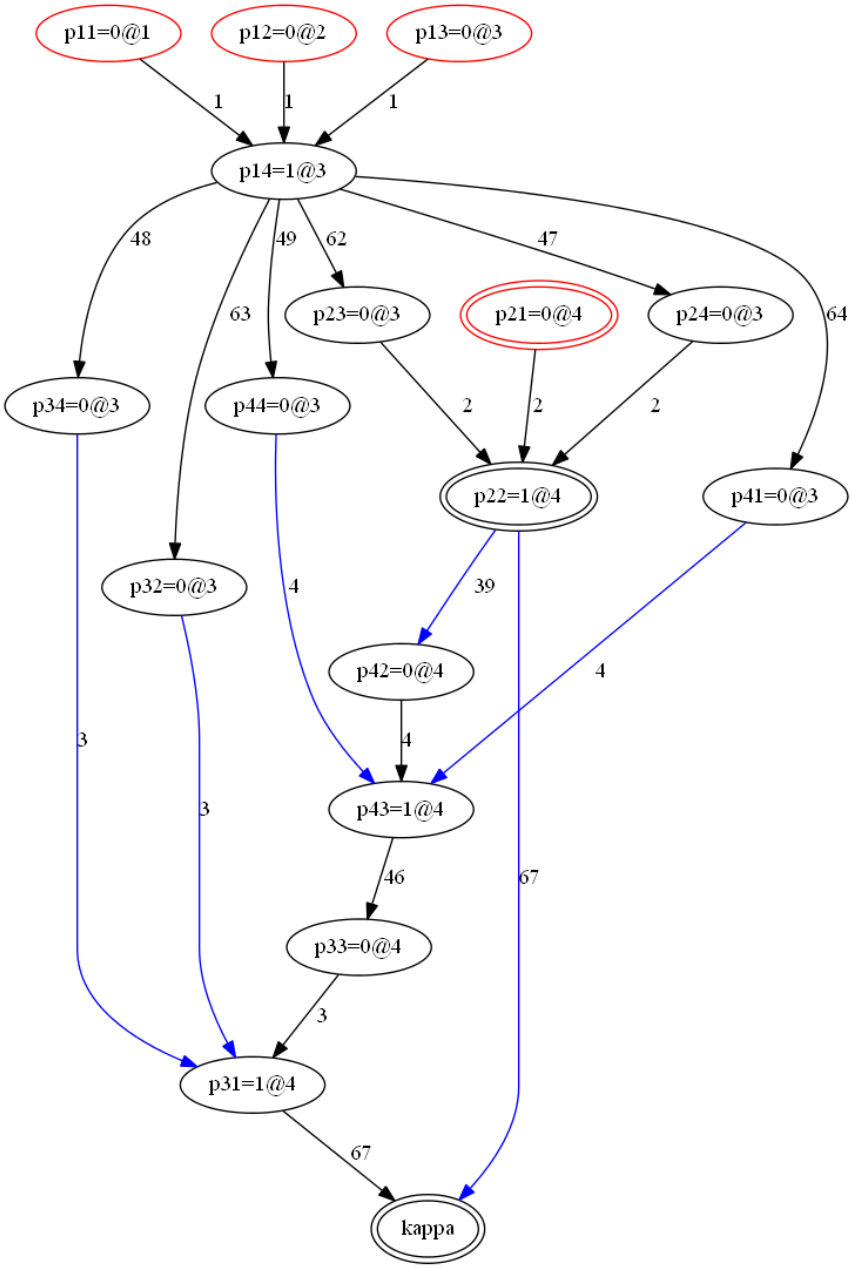
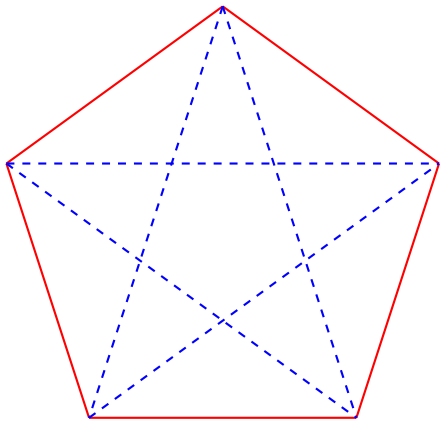


Figure 4.4: Implication graph for the four-queens

4.4 Ramsey numbers

File: `ramsey.pro`

Ramsey's Theorem states that for any n, there is a smallest number $R(n)$ with the following property: If the edges of the complete graph on $R(n)$ vertices are colored with two colors then there must be a monochromatic complete subgraph with n vertices. The archive contains CNF formulas that show that $R(3) > 5$ and $R(3) = 6$. The following drawing shows that it is possible to color K_5 with two colors such that there is no monochromatic triangle.



The CNF encoding uses `true` for one color and `false` for the other. For each of the ten triangles in K_5, there is a pair of clauses that ensure that at least one edge is colored `true` and one `false`:

```
[x12,x23,x13], [~x12,~x23,~x13],
```

The set of clauses is satisfiable.

For K_6, there is pair of clauses for each of the nineteen triangles. The formula is unsatisfiable, meaning that it is not true that there exists a triangle that is not monochromatic, that is, there must be a monochromatic triangle. Therefore, $R(3) = 6$.

4.5 Schur triples

File: `schur.pro`

Given *any* decomposition of the natural numbers $S = \{1,\ldots,n\}$ into two mutually exclusive subsets S_1, S_2, do there exist $a,b,c \in S_i$, $a < b < c$, $a + b = c$, for at least one $i = 1,2$? The set of numbers $\{a,b,c\}$ is called a *Schur triple*. For $n = 8$, there are decompositions with no triples:

$$S_1 = \{1,2,4,8\}, S_2 = \{3,5,6,7\}.$$

The archive contains CNF formulas showing that for $n = 8$, there are decompositions with no Schur triples, but for $n = 9$, all decompositions contain a Schur triple. If equality is allowed $a \leq b$, the numbers are $n = 4$ and $n = 5$, respectively.

The encoding is similar to that used for Ramsey numbers with `true` representing one set and `false` representing the other set. For each $\{a,b,c\}$ such that $a+b=c$, the following pair of clauses states that at least one of x_a, x_b, x_c must be `true` and at least one must be `false`:

```
[x3,x4,x7], [~x3,~x4,~x7],
```

For $n = 8$ the formula is satifiable, but for $n = 9$ it is unsatisfiable.

4.6 Langford's problem

File: `generate-langford.pro, langford.pro`

Langford's problem $L(n)$ asks if it is possible to arrange pairs of numbers $1,1,2,2,3,3,\ldots,n,n$ in a sequence so the two occurrences of each number k are separated by k other numbers. It is easy to see that 312132 is a solution for $n = 3$ and 41312432 is a solution for $n = 4$. Here we show how to solve $L(3)$ and $L(4)$ using SAT solving. The presentation is based on [7]; for more information on Langford's problem see [9].

The problem can be posed using an array. For $n = 3$, there are 6 columns, one for each of the $2n$ numbers. The rows represent the specification of the problem: the occurrences of k must have k numbers between them. It is easy to see that there are four possible placements of 1, three of 2 and two of 3:

	1	2	3	4	5	6
1	1		1			
2		1		1		
3			1		1	
4				1		1
5	2			2		
6		2			2	
7			2			2
8	3				3	
9		3				3

To solve the problem, we need to select *one* row for the positions of the 1's in the sequence, *one* row for the 2's and *one* row for the 3's, such that if we stack just these rows on top of each other, no column contains more than one number:

	1	2	3	4	5	6
2		1		1		
7			2			2
8	3				3	

First, note that row 9 is not needed because of symmetry: starting with row 9 just gives the reversal of the sequence obtained by starting with row 8.

Row 8 is the only one containing 3's so it must be chosen and the result is $3\_\_\_3\_$. Any row with numbers in columns 1 and 5 can no longer be used, because only one number can be placed at each position: $\not{1},2,\not{3},4,\not{5},\not{6},7,8$.

Row 7 is the only remaining row containing 2's so must be chosen and the result is 3_2_32. Deleting rows that can no longer be used gives: $\not{1},2,\not{3},\not{4},\not{5},\not{6},7,8$.

Choosing the only remaining row, row 2, gives 312132.

Let us encode Langford's problem. Let x_i be true if row i is chosen. The following clauses encode that exactly one of rows 1,2,3,4 must be chosen:

```
[x1,x2,x3,x4],
[~x1,~x2], [~x1,~x3], [~x1,~x4],
[~x2,~x3], [~x2,~x4],
[~x3,~x4]
```

The first clause encodes that at least one of these rows must be chosen and the other clauses encode at most one of the rows can be chosen. There are similar sets of clauses for rows 5, 6, 7, and for row 8 (which is a unit clause because there are no alternatives).

In the same way, we encode clauses expressing the constraints on the columns. For example, column 1 requires that exactly one of rows 1, 5, 8 be chosen:

```
[x1,x5,x8], [~x1,~x5], [~x1,~x8], [~x5,~x8]
```

For $n = 3$ there are 32 clauses on the 8 variables, and the DPLL algorithms finds the solution immediately using just unit propagation starting with `x8`.

The archive contains a set of clauses $L(4)$. Running DPLL gives:

```
clauses=101, variables=17, units=50, decisions=10, conflicts=4
```

The reader is encouraged to check if `cdcl`, `ncb` or `look` can improve the performance.

4.7 van der Waerden's problem

File: `generate-vdw.pro, vdw.pro`

Consider two sequences of eight binary digits:

$$00011100,\ 01010011,\ 00110011\,.$$

The first sequence has a subsequence of three consecutive 0's, as well as a one of three consecutive 1's. The second sequence has three 0's two positions away from each other: the digits at position 1 (counting from the left), position 3 and position 5. The third sequence has *no* subsequence of three equal digits that equally spaced from each other. If you try to find a sequence of *nine* binary digits with no subsequence of equally spaced digits, you will not succeed. For example, adding a 0 to the end of the third sequence above gives 001100110 and there is a subsequence of 0's at positions $1,5,9$, and if add a 1, the sequence 001100111 has three consecutive 1's at the end.

van der Waerden's problem for k asks what is the smallest number n such that any sequence of n digits contains k equally spaced digits.

To solve van der Waerden's problem with a SAT solver, use an encoding similar to the encoding of Schur triples. For $n = 8$ the encoding is:

```
[x1,x2,x3],[~x1,~x2,~x3],
[x2,x3,x4],[~x2,~x3,~x4],
[x3,x4,x5],[~x3,~x4,~x5],
[x4,x5,x6],[~x4,~x5,~x6],
[x5,x6,x7],[~x5,~x6,~x7],
[x6,x7,x8],[~x6,~x7,~x8],
[x1,x3,x5],[~x1,~x3,~x5],
[x2,x4,x6],[~x2,~x4,~x6],
[x3,x5,x7],[~x3,~x5,~x7],
[x4,x6,x8],[~x4,~x6,~x8],
[x1,x4,x7],[~x1,~x4,~x7],
[x2,x5,x8],[~x2,~x5,~x8]
```

This formula is satisfiable:

```
Satisfying assignments: [x1=0,x2=0,x3=1,x4=1,x5=0,x6=0,x7=1,x8=1]
```

and corresponds to the sequence 00110011 which has no equally spaced subsequence of length three. We can extending the encoding for sequences of length $n = 9$ by adding the clauses:

```
[x7,x8,x9],[~x7,~x8,~x9],
[x5,x7,x9],[~x5,~x7,~x9],
[x3,x6,x9],[~x3,~x6,~x9],
[x1,x5,x9],[~x1,~x5,~x9]
```

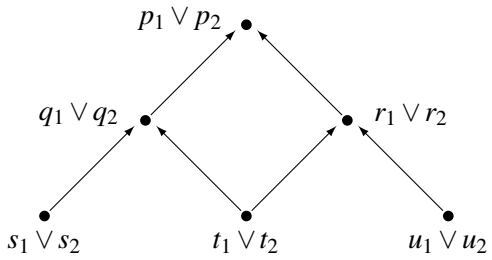
LEARNSAT now returns that the formula is unsatisfiable, meaning that there is no sequence without an equally spaced subsequence of length three.

The archive contains formulas that show that there are sequences of length $n = 34$ with no equally spaced subsequences of length four, but that is not possible for sequences of length $n = 35$. All formulas were generated by the program `generate-vdw.pro`.

4.8 Pebbling formulas

`File: pebbling.pro`

A pebbling formula represents the scheduling of tasks. Each task is represented by a node in a directed acylic graph. In the following diagram the task represented by the topmost node can be performed by "machine" p_1 *or* by "machine" p_2; however, it can begin only when both of the two tasks below it have been completed. In turn, each of those tasks can be performed by one of two machines and can begin only after the previous tasks at the source of its incoming edges have been completed.



The pebbling formula for a graph is composed of the following subformulas: (a) For each node, there is a disjunction of the atoms representing the machines that can carry out the task at that node. (b) The scheduling constraints for each non-leaf node are represented as implications; for the topmost node this is:

$$(q_1 \vee q_2) \wedge (r_1 \vee r_2) \rightarrow (p_1 \vee p_2),$$

which in clausal form is:

$$\{\{\neg q_1, \neg r_1, p_1, p_2\}, \{\neg q_1, \neg r_2, p_1, p_2\}, \{\neg q_2, \neg r_1, p_1, p_2\}, \{\neg q_2, \neg r_2, p_1, p_2\}\}.$$

Clearly, the union of these sets of clauses is satisfiable simply by assigning true to each atom. However, if we add (c) the formula $\neg p_1 \wedge \neg p_2$ (the two unit clauses $\neg p_1$ and $\neg p_2$), the formula becomes unsatisfiable.

The formula associated with the graph shown in the diagram is called the 3-layer *grid pebbling formula*; similar formula can be given for any n. Even for the small 3-layer formula, clause learning makes a significant difference:

```
dpll: units=74, decisions=50, conflicts=26
cdcl: units=25, decisions=14, conflicts=8, learned clauses=5
ncb:  units=17, decisions=9,  conflicts=3, learned clauses=3
```

Let us trace the computation in mode NCB. The clauses $\{\neg p_1\}$ and $\{\neg p_2\}$ are unit clauses and can be immediately propagated:

```
Propagate unit: ~p1 (p1=0@0) derived from: [~p1]
Propagate unit: ~p2 (p2=0@0) derived from: [~p2]
```

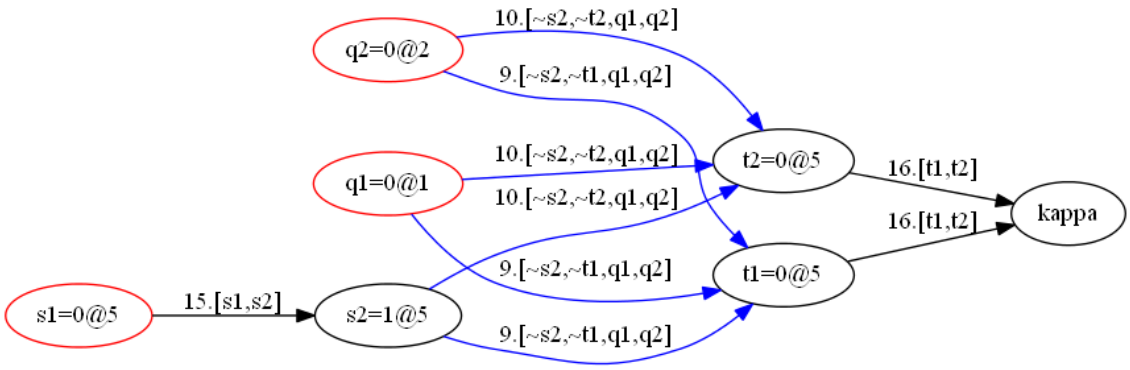
Next decision assignments are made for the four variables at the second layer:

```
Decision assignment: q1=0@1
Decision assignment: q2=0@2
Decision assignment: r1=0@3
Decision assignment: r2=0@4
```

When a decision is made for a variable at the third layer, additional unit propagations lead to a clause associated with a leaf becoming a conflict clause:

```
Decision assignment: s1=0@5
Propagate unit:  s2 (s2=1@5) derived from: [s1,s2]
Propagate unit: ~t1 (t1=0@5) derived from: [~s2,~t1,q1,q2]
Propagate unit: ~t2 (t2=0@5) derived from: [~s2,~t2,q1,q2]
Conflict clause: 16. [t1,t2]
```

From the implication graph:



we see that the learned clause is:

```
Learned clause from resolution: [~s2,q1,q2]
```

After learning the clause, three levels of decision assignments are skipped:

```
Non-chronological backtracking to level: 2
Writing dot graph: 0
Skip decision assignment: s1=1@5
Skip decision assignment: r2=1@4
Skip decision assignment: r1=1@3
```

(Clearly, the decision assignments for r_1 and r_2 are irrelevant to the conflict.)

The subsequent decision assignments and unit propagation result in the same conflict clause and a new learned clause with backtracking again to level 2:

```
Conflict clause: [u1,u2]
Learned clause from resolution: [~t2,r1,r2]
Non-chronological backtracking to level: 2
```

The same conflict appears again, but after learning the clause, the set of clauses is immediately determined to be unsatisfiable because there are no longer any decisions that have not been made:

```
Conflict clause: [u1,u2]
Learned clause from resolution: [~q1,p1,p2]
Unsatisfiable:
```

The variable ordering is extremely important here. The default ordering is lexicographic and we happened to put the first variables at the apex of the graph. If we change the variable ordering to start with the leaves and work up the graph, proving the formula unsatisfiable is much less efficient even for mode NCB:

```
ncb:   units=26, decisions=21, conflicts=9
```

4.9 Seating guests at a table

These examples were contributed by Na'ama Tal and Nili Gilboa Luria.

File: `tablea.pro, tableb.pro, tablec.pro`

Solving problems in the presence of constraints is an important application of SAT solving because constraints are easily expressed as logical formulas. The example we take is to compute a plan for seating guests at a round table, where the constraints are of one of two forms: Guest g_i *must* sit next to guest g_j or guest g_i *refuses* to sit next to guest g_j. The examples will use a table with six seats for six guests.

4.9.1 Encoding the problem

The atomic proposition `pij` is true iff guest g_i sits in chair c_j, $1 \leq i, j \leq 6$.
Each guest must sit in some chair, so we have six clauses of the form:

```
[pi1, pi2, pi3, pi4, pi5, pi6]
```

There is one clause for each $1 \leq i \leq 6$.
Similarly, no more than one guest can sit in any chair, so we have six sets of clauses of the form:

```
[~p1j, ~p2j], [~p1j, ~p3j], [~p1j, ~p4j], [~p1j, ~p5j], [~p1j, ~p6j],
[~p2j, ~p3j], [~p2j, ~p4j], [~p2j, ~p5j], [~p2j, ~p6j],
[~p3j, ~p4j], [~p3j, ~p5j], [~p3j, ~p6j],
[~p4j, ~p5j], [~p4j, ~p6j],
[~p5j, ~p6j]
```

There is one set of clauses for each $1 \leq j \leq 6$.
To encode that g_i refuses sit next to guest g_j, use a clause for each of the two adjacent chairs c_{k-1}, c_{k+1} (where addition are subtraction are modulo 6):

$$p_{ik} \rightarrow \neg p_{j(k-1)} \wedge \neg p_{j(k+1)} \equiv (\neg p_{ik} \vee \neg p_{j(k-1)}) \wedge (\neg p_{ik} \vee \neg p_{j(k+1)})$$

For example, if g_1 refuses to sit next to g_2, then if g_1 sits in c_1, g_2 can't sit in c_2 or c_6:

```
[~p11, ~p22], [~p11, ~p26]
```

Each constraint gives rise to six pairs of clauses, one for each chair.
There are two ways to encode the constraint that guest g_i must sit next to guest g_j. One way is to take implications of the form:

$$p_{ik_1} \rightarrow \neg p_{jk_2} \equiv \neg p_{ik_1} \vee \neg p_{jk_2}$$

for every pair of chairs k_1, k_2 that are *not* adjacent. For example, if g_3 must sit next to g_4, then if g_3 sits in c_1, g_4 must sit in c_6 or c_2, *not* in c_3, c_4 or c_5:

```
[~p31, ~p43], [~p31, ~p44], [~p31, ~p45]
```

There will be six sets of three clauses, one set for each chair.
The second way is to directly encode the adjacency requirements; if g_i sits in c_k then g_j must sit in c_{k-1} or c_{k+1}:

$$p_{ik} \rightarrow p_{j(k-1)} \vee p_{j(k+1)} \equiv \neg p_{ik} \vee p_{j(k-1)} \vee p_{j(k+1)}.$$

In the example, if g_3 must sit next to g_4, then if g_3 sits in c_1, g_4 must sit in c_2 or c_6:

```
[~p31, p42, p46]
```

There is one three-literal clause for each of the six chairs.

4.9.2 Three problems

The archive contains three problems for seating guests.
File `tablea.pro` encodes the constraints:

- g_1 refuses to sit next to g_2
- g_4 refuses to sit next to g_5
- g_3 must sit next to g_4
- g_3 must sit next to g_6

and the first encoding of g_i must sit next to guest g_j is used. The problem is satisfiable under the assignment:

```
p16=1, p24=1, p32=1, p43=1, p55=1, p61=1
```

and is solved in *all* modes with the statistics:

```
units=44, decisions=15, conflicts=2, learned clauses=0
```

File `tableb.pro` encodes the constraints:

- g_1 refuses to sit next to g_2
- g_4 refuses to sit next to g_5
- g_3 must sit next to g_1
- g_3 must sit next to g_4
- g_3 must sit next to g_6

and the second, more efficient, encoding of g_i must sit next to guest g_j is used. The problem is unsatisfiable but there is still no difference in the statistics in the different modes:

```
units=759, decisions=120, conflicts=61, learned clauses=5
```

File `tablec.pro` encodes the constraints:

- g_1 refuses to sit next to g_2
- g_3 must sit next to g_4
- g_4 must sit next to g_5

- g_5 must sit next to g_6

and the efficient encoding is used. This time, the three modes give significantly different results:

```
dpll: units=1187, decisions=286, conflicts=144
cdcl: units=1016, decisions=232, conflicts=117, learned clauses=26
ncb : units=926,  decisions=207, conflicts=92,  learned clauses=14
```

The examples show that it is hard to predict when the advanced algorithms improve performance.

4.10 Graph coloring

File: `mcgregor.pro`

One of the most famous mathematical problems concerns the coloring of planar graphs, often called maps, such that no adjacent areas receive the same color.[1] In 1976 was the Four Color Theorem proved that four colors are sufficient. Shortly before the proof appeared, Martin Gardner claimed *five* colors were needed to color a map called the *McGregor graph of order* 10. This was one of a series of spoofs for April Fools day, though many reader were taken in.

The archive contains programs that use SAT solving to show that the McGregor graph of order 3 (Figure 4.5) cannot be colored with three colors but can be colored with four colors.

The map has twelve areas. For each area, a clause is needed to ensure that the area is colored by at least one color, and a set of clauses is needed to require that adjacent areas are colored with different colors. Suppose that we are trying to find a four-coloring of the graph. The variables are `vnc`, where `n` is the label of an area and `c` is one of the colors. For area 22 in the graph, there is a clause:

```
[v201,v202,v203,v204]
```

requiring that at least one of the colors 1, 2, 3, 4 be assigned to that area. The following clauses require that color 1 cannot be used *both* for area 20 *and* for one of 21, 22, 30, 31:

```
[~v201,~v211], [~v201,~v221], [~v201,~v301], [~v201,~v311]
```

the areas adjacent to area 20. There are similar sets of clauses for colors 2, 3 and 4. Since we generated the clauses in ascending order of the labels of the areas, we do not have to include clauses like `[~v201,~v111]`, because `[~v111,~v211]` was previously included and the clauses are commutative.

Run the program and check that the variables assigned 1 define a correct four-coloring.

[1] This section is based on [7, pp. 7, 134]

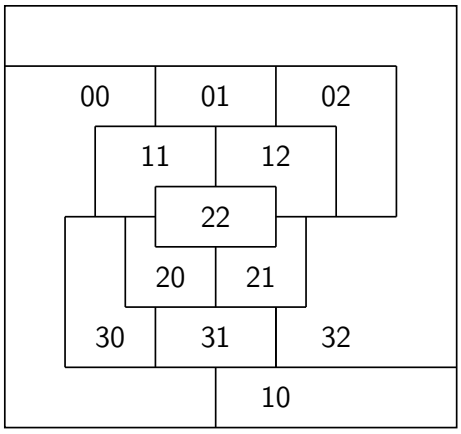


Figure 4.5: McGregor graph of order 3

4.11 Colored queens

File: `color-queens.pro`

Can you place k colored queens on every square of an $n \times n$ board such that no color appears more than once in any row, column or diagonal? For a 4×4 board, the left diagram below shows a failed attempt to assign four colors (A,B,C,D). Although no color appears more than once in any row or column, colors appear multiple times on the diagonals. In fact, there is no solution to the problem for four colors, but there are solutions for five colors (A,B,C,D,E) as seen in the right diagram:

A	D	C	B
B	C	D	A
C	B	A	D
D	A	B	C

E	D	C	B
C	B	A	E
A	E	D	C
D	C	B	A

Let the variables be `pijk` where `i`, `j` are the row and column of a square and `k` is the color assigned to that square. The following clauses require that each of the squares in the first row contain (at least) one of four colors:

```
[p111, p112, p113, p114],
[p121, p122, p123, p124],
[p131, p132, p133, p134],
[p141, p142, p143, p144]
```

Similar clauses are needed for the other rows.

For each square, we need *exclusion clauses* which require that if a color appears in a square then it does not appear in any square in the same row, column and diagonal. Here are the exclusion clauses for `p111`, the literal that claims that the first square is colored `1`:

```
[~p111, ~p121], [~p111, ~p131], [~p111, ~p141],
[~p111, ~p211], [~p111, ~p311], [~p111, ~p411],
[~p111, ~p221], [~p111, ~p331], [~p111, ~p441]
```

There are three more sets of these clauses for the other colors.

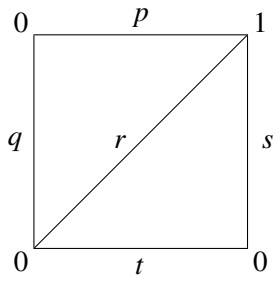
As you construct exclusion clauses for each successive square, their number decreases since the requirements are commutative. For example, once you have added `[~p113,~p123]` to require that squares `11` and `12` not both be colored with color `3`, you don't need `[~p123,~p113]`.

Write sets of clauses to prove the obvious facts that you can't color a 2×2 board with two or three colors, but you can with four colors.

4.12 Tseitin graphs

File: `GenerateSatPRO.java, tseitin.pro`

G.S. Tseitin used formulas associated with graphs in his research on the complexity of resolution.[2] Consider the following connected undirected graph whose vertices are labeled with 0 or 1 and whose edges are labeled with letters:



The formula associated with this graph is defined as follows: for each vertex, construct the set of clauses whose atoms are the letters labeling the incident edges, such that the parity (sum modulo 2) of the number of negated literals is not equal to the label of the vertex:

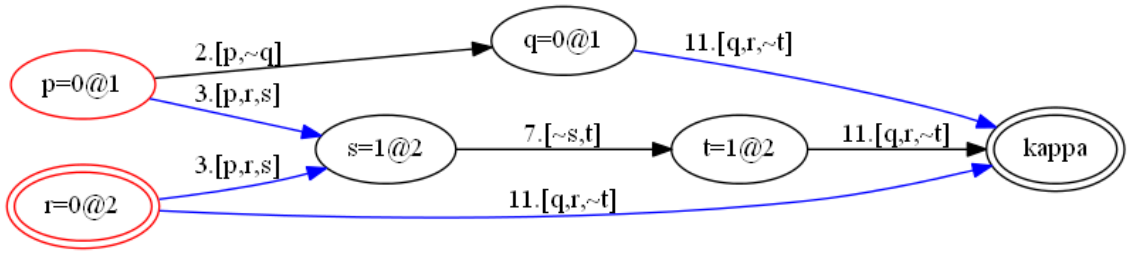
```
[~p, q], [p, ~q],
[p, r, s], [~p, ~r, s], [~p, r, ~s], [p, ~r, ~s],
[~s, t], [s, ~t],
[~q, r, t], [q, ~r, t], [q, r, ~t], [~q, ~r, ~t]
```

[2] See [2, Section 4.5] for a formal presentation of Tseitin formulas.

For example, since $\{p,r,s\}$ are incident with a vertex labeled 1, the number of negated literals in a cluase on these atoms has to be 0 or 2.

If the number of vertices labeled 1 is odd, the formula is unsatisfiable.

Let us look at the implication graph:



The only dominator of the lowest decision node `r=0@2` is the decision node itself. Since it is not implied by other decision node, we cannot skip it in backtracking and the assignment `r=1@2` will have to be tried.

Another anomaly of this example is that the clause that would be learned from the dominator is not the same as the one learned by resolution. The former is `[p,r]`, which is the complement of the dominator decision node `r=0@2` and the decision node at a higher level `p=0@1`. The clause learned by resolution is obtained as follows:

```
Conflict clause: 11. [q,r,~t]
Resolvent: of [q,r,~t] and antecedent [~s,t] is [q,r,~s]
Resolvent: of [q,r,~s] and antecedent [p,r,s] is [q,p,r]
UIP: one literal r is assigned at level: 2
Learned clause from resolution: [q,p,r]
```

This is a weaker clause than that defined by the dominator.

The archive includes programs for Tseitin clauses on the bipartite graphs $K_{2,2}, K_{3,3}, K_{4,4}$. These were generated by the Java program `GenerateSatPRO.java`.

Chapter 5

Bounded model checking

`File: bmc-generate-sem.pro, bmc-sem.pro`

Model checking a method for verifying the correctness of computer hardware and software. A model checker inputs a formal description of a system and a correctness claim and then searches for states in the execution of the system that falsify the claim.

In *explicit-state* model checkers, the reachable states of the system are generated one-by-one and the correctness claim is evaluated after each new state is evaluated. If the system has a finite number of states, eventually the model checker will either find a state falsifying the claim or declare that the claim is true in all reachable states. Clever algorithms and optimized data structures ensure that many real systems can be completely checked. For an overview of explicit-state model checking (in particular, the SPIN model checker), see [1].

Symbolic model checkers represent the set of reachable states and a correctness claim in a data structure and then perform computations on this data structure, leading to a falsifying state or to a determination that the correctness claim holds. In *bounded model checking*, the system and (the negation of) its correctness claim are represented by a formula in propositional logic. If a SAT solver finds that the formula is satisfiable, the exists a state that does not fulfill the correctness claim. The formula has subformulas for each step in the computation. Since a SAT solver expects to receive a finite formula, it is necessary to limit *a priori* the number of execution steps, hence *bounded* model checking. Fortunately, it is usually possibly to determine a bound such that if the system is not correct, a state that falsifies the correctness claim will be found within that bound. See [3] for a survey of bounded model checking.

5.1 Encoding a concurrent program as a SAT problem

Consider the solution of the critical section problem using a semaphore:

s: semaphore := 1	
process p	process q
non-critical section	con-critical section
wait(s)	wait(s)
critical section	critical section
signal(s)	signal(s)

There are four statements in each process, but the only statements that affect the synchronization are the semaphore operations. The algorithm can be represented as follows, where we agree that a process is in its critical section if it is about to execute the signal operation:

s: semaphore := 1	
process p	process q
wait(s)	wait(s)
signal(s)	signal(s)

Each state can be represented by an atom composed of three letters: `w` or `s`—meaning *at wait* or *at signal*—for each of the two processes, and `o` or `z`—meaning that the value of `s` is one or zero. In addition, each atom will have a number: `0`, `1`, `2`, `3`, ..., for the steps of the execution. The initial state is `wwo0`. The state `ssz5` means that after step 5 both processes are at their signal instructions and the value of the semaphore is zero. Of course, this means that mutual exclusion does not hold so the program is not correct.

Notation: `^` is conjunction, `<->` is equivalence, `+` is exclusive or, `N` is a step, `N'` is the next step.

The initial state requires that the following formula be true:

```
(wwo0 ^ ~wwz0 ^ ~swo0 ^ ~swz0 ^ ~wso0 ^ ~wsz0 ^ ~sso0 ^ ~ssz0)
```

The transitions of the system are represented by the following formula:

```
((wwoN <-> swzN') + (wwoN <-> wszN')) ^
(swoN <-> sszN') ^
(swzN <-> wwoN') ^
(wsoN <-> sszN') ^
(wszN <-> wwoN') ^
((sszN <-> wsoN') + (sszN <-> swoN'))
```

The first formula says that in state `wwo`, *either* process `p` executes the wait instruction and reduces the value of `s` to 0 *or* process `q` executes the wait instruction and reduces the value of `s` to 0, *but not both* (because we are using an interleaving semantics of concurrency).[1]

Let us now guess that if there is a state that does not fulfill mutual exclusion, it will occur within two steps. The formula representing the computation consists of the conjunction of the formula for the initial state and two copies of the formula for the transitions, one with `N` equal to 0 and `N'` equal to 1 and another with `N` equal to 1 and `N'` equal to 2.

Finally, take the conjunction of that formula together with the formula representing the negation of the correctness claim: after 0, 1 or 2 steps, both processes are in their critical sections:[2]

```
(ssz0 v sso0  v  ssz1 v sso1  v  ssz2 v sso2)
```

The resulting formula is satisfiable if and only if mutual exclusion does not hold within two steps.

The program `bmc-sem.pro` contains a predicate `generate` that takes the above formula, converts it to CNF and writes (on file `bmc.pro`) the set of clauses in the correct form for running LEARNSAT.[3]

5.2 Running the SAT solver on the encoding

Load the file `bmc.pro`-sem in the PROLOG compiler and run the predicate `bmc`. The result is that the formula is unsatisfiable, meaning that a violation of mutual exclusion does not happen within the first two steps. Of course, it is possible that it occurs after step 3, or step 4, or step 100, but from the structure of the program that seems unlikely.

The statistics for the three different modes are:

```
dpll: units=25, decisions=12, conflicts=7
cdcl: units=19, decisions=4,  conflicts=3, learned clauses=2
ncb:  units=16, decisions=3,  conflicts=2, learned clauses=2
```

showing that the advanced algorithms are more efficient.

However, let us examine the trace in more detail. Since the original set of clauses contains many unit clauses, the execution starts with a series of unit propagations:

```
Propagate unit:  wwo0 (wwo0=1) derived from: [wwo0]
Propagate unit: ~wwz0 (wwz0=0) derived from: [~wwz0]
Propagate unit: ~swo0 (swo0=0) derived from: [~swo0]
```

[1] We assume a binary semaphore with values 0 and 1 only, so signal cannot be executed if the value of `s` is 1.

[2] The value of the semaphore is irrelevant.

[3] The program uses files `cnf.pro` and `ops.pro` that were adapted from the program archive that accompanies [2].

```
Propagate unit: ~swz0 (swz0=0) derived from: [~swz0]
Propagate unit: ~wso0 (wso0=0) derived from: [~wso0]
Propagate unit: ~wsz0 (wsz0=0) derived from: [~wsz0]
Propagate unit: ~sso0 (sso0=0) derived from: [~sso0]
Propagate unit: ~ssz0 (ssz0=0) derived from: [~ssz0]
Propagate unit: ~ssz1 (ssz1=0) derived from: [swo0,~ssz1]
Propagate unit: ~wwo1 (wwo1=0) derived from: [swz0,~wwo1]
```

Next, two decision assignments are made:

```
Decision assignment: sso1=0
Decision assignment: sso2=0
```

but those seem strange. The initial state `wwo0` is one in which both processes are at their wait operations at step 0, so why does the algorithm start by assigning values to atoms associated with signal operations at steps 1 and 2? By default, the algorithm makes decisions assignment in lexicographic order of the atoms:[4]

```
Variables: [sso0,sso1,sso2,ssz0,ssz1,ssz2,swo0,swo1,swo2,swz0,swz1,swz2,
wso0,wso1,wso2,wsz0,wsz1,wsz2,wwo0,wwo1,wwo2,wwz0]
```

By following our intuition, let us change the order of the atoms by putting atoms in the second line before those in the first line so that those beginning with `w` are first:

```
set_order(
    [wso0,wso1,wso2,wsz0,wsz1,wsz2,wwo0,wwo1,wwo2,wwz0,
     sso0,sso1,sso2,ssz0,ssz1,ssz2,swo0,swo1,swo2,swz0,swz1,swz2]).
```

Now, the algorithm in dpll mode is more efficient than it was in ncb mode!

```
dpll: units=14, decisions=2, conflicts=2
```

This demonstrates that there is an element of nondeterminism in these algorithms for SAT solving. Since it would not be tractable to try all permutations of the order of the variables, there remains an element of "luck" when you choose an ordering arbitrarily, at random or using heuristics.

[4] This list can be displayed by selecting display option `variable`.

Bibliography

[1] M. Ben-Ari. A primer on model checking. *ACM Inroads*, 1(1):40–47.

[2] M. Ben-Ari. *Mathematical Logic for Computer Science (Third Edition)*. Springer, 2012.

[3] A. Biere. *Bounded Model Checking*, chapter 14, pages 457–481. In Biere et al. [4], 2009.

[4] A. Biere, M. Heule, H. Van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*. IOS Press, 2009.

[5] Marijn J. H. Heule and Oliver Kullmann. The science of brute force. *Commun. ACM*, 60(8):70–79, 2017.

[6] Marijn J. H. Heule and van Maaren, H. *Look-Ahead Based SAT Solvers*, chapter 5, pages 155–184. In Biere et al. [4], 2009.

[7] D.E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 6: Satisfiability*. Pearson, 2015.

[8] J. P. Marques-Silva, I. Lynce, and S. Malik. *Conflict-Driven Clause Learning SAT Solvers*, chapter 4, pages 131–153. In Biere et al. [4], 2009.

[9] J.E. Miller. Langford's Problem. `http://dialectrix.com/langford.html`.

[10] B.A. Nadel. Representation selection for constraint satisfaction: A case study using n-queens. *IEEE Expert: Intelligent Systems and Their Applications*, 5:16–23, June 1990.

[11] S. Prestwich. *CNF Encodings*, chapter 2, pages 75–97. In Biere et al. [4], 2009.