number of different techniques for solving the SAT problem, some of which are dramatically more efficient than others, exhibiting remarkably good performance even for very large and structurally complex sentences. However, given that all of these problems are paradigmatically NP-complete, we cannot expect any algorithm to have guaranteed good (polynomial-time) performance on all possible inputs—at least not unless $P = NP$! Even the most efficient SAT solver will suffer degraded performance on some inputs. We study techniques for solving the SAT problem in the next section.

Earlier in this chapter, on page 198, we wrote that a tautology is "a sentence that can be *derived* from every assumption base." That was a proof-centered or syntactic characterization of tautologies. We mentioned in footnote 3 that the syntactic characterization coincides with the semantic one that we developed in this section because Athena's proof system is *complete*. A more precise formulation of completeness is given by Theorem 4.8 below. A similar result is given for a closely related (and arguably more important) property, *soundness*. We do not prove these results here. Soundness could be established by induction on the structure of Athena deductions. Proving completeness is a bit more challenging, but it could be done either directly, using techniques that go back to Henkin [46], or by reduction to a proof system that is already known to be complete, for example, by showing that every result that can be derived in that system can also be derived in Athena's sentential logic.

Theorem 4.7: Soundness

If a deduction D produces a conclusion p in an assumption base β, then $\beta \models p$.

Theorem 4.8: Completeness

If $\beta \models p$ then there is a deduction D that derives p from β.

4.13 SAT solving

The most naive way of solving the tautology problem (and hence also SAT) for a given p is the enumeration technique mentioned in the previous section: Generate all 2^n interpretations that distribute truth values to the n atoms of p in different ways and check that p comes out `true` under each of these interpretations. (Note: The library procedure `atoms` returns a (deduped) list of all the atoms that occur in a given sentence.)

```
define (all-interpretations atoms) :=
  match atoms {
    [] => [|{}|]
  | (list-of a t) => let {L := (all-interpretations t)}
                       (join (map lambda (I) (Map.add I [[a true]]) L)
                             (map lambda (I) (Map.add I [[a false]]) L))
```

```
   }

define (taut? p) :=
  (for-each (all-interpretations (atoms p)) lambda (I) (V p I))

set-precedence taut? 5

> (taut? A & B ==> B & A)

Term: true

> (taut? (A & B <==> C | D) ==> ~ C ==> ~ D ==> ~ A | ~ B)

Term: true

> (taut? A & B)

Term: false
```

We can now define a procedure `sat?` for determining satisfiability directly in terms of `taut?` by taking advantage of Theorem 4.1:

```
define (sat? p) := (negate taut? ~ p)

set-precedence sat? 5

> (sat? A & B)

Term: true

> (sat? A & ~ A)

Term: false
```

However, as mentioned earlier, for satisfiable sentences we are typically not content with a "yes" answer—we also want a satisfying interpretation (and sometimes several or all satisfying interpretations). The following implementation of `sat?` returns an interpretation (as a map) if there is one, otherwise it returns `false`.

```
define (sat? p) :=
 (find-element (all-interpretations atoms p)
               lambda (I) (V p I)
               lambda (x) x
               lambda () false)
set-precedence sat? 5
```

The generic list procedure `find-element` has the following definition:

```
define (find-element L pred success failure) :=
  letrec {loop := lambda (L)
                    match L {
                      [] => (failure)
                    | (list-of h t) => check {(pred h) => (success h)
                                            |  else => (loop t)}
                    }}
    (loop L)
```

We now have:

```
> (sat? A & B)

Map: |{A := true, B := true}|

> (sat? A & ~ B)

Map: |{A := true, B := false}|

> (sat? A & ~ A)

Term: false
```

While this approach works well enough for small inputs, its exponential complexity quickly becomes prohibitive.

```
declare P: [Int] -> Boolean

define (if-chain N) :=
  let {P_1 := (P 1);
       P_N := (P N);
       conditionals := (map lambda (i) (P i ==> P i plus 1)
                            (1 to N minus 1));
       antecedent := (& (P_1 added-to conditionals))}
   (antecedent ==> P_N)

define p := (if-chain 3)

> p

Sentence: (if (and (P 1)
                   (if (P 1)
                       (P 2))
                   (if (P 2)
                       (P 3)))
              (P 3))
```

Clearly, `(if-chain` n`)` is a tautology for every $n > 1$. Even for $n = 20$,

`(taut? if-chain` n`)`

takes a while to finish. After all, the algorithm has to generate and then check more than 1 million interpretations ($2^{20} = 1048576$)! For larger values yet, the combinatorics become hopeless. By contrast, we will see that state-of-the-art SAT solvers solve instances of `(if-chain n)` instantly, even for values of n exceeding 10^6. (More precisely, they solve instances of `(~ if-chain n)`, recognizing that such sentences are unsatisfiable, which, by Theorem 4.1, amounts to `(if-chain n)` being a tautology.)

Modern SAT solving is based on the *DPLL procedure*, an acronym for the "Davis-Putnam-Logemann-Loveland" algorithm invented in the late 1950s by Martin Davis, Putnam, Hilary, Loveland, Donald, and Logemann, George.[21] In what follows we will implement the core DPLL algorithm in Athena. DPLL expects its input formula p to be in CNF (conjunctive normal form), meaning that p must be a conjunction of disjunctions of literals, where a literal is either an atom or the negation of an atom. See Exercise 4.32 for additional discussion of this normal form and for algorithms that can transform arbitrary sentences into CNF. We will use Athena's built-in CNF conversion procedure later to test our implementation of DPLL.

We represent a disjunction d of literals as a list L_d of those literals, with the understanding that the order of the literals in the list is immaterial (in view of the commutativity and associativity of disjunction), and we represent a conjunction of such disjunctions by a list of the underlying disjunctive lists. So, for instance, the CNF formula

```
((A | B) & (B |~ C) & D)
```

might be represented by the list

```
[[A B] [B (~ C)] [D]].
```

A list of this kind will be the main input to our DPLL implementation. The output will be either the term `'unsat`, indicating that the input is unsatisfiable, or else an interpretation, represented simply as a list of literals. Observe that an empty list of clauses is trivially satisfiable (every interpretation satisfies it); and that an empty clause `[]` is tantamount to `false`. To see the latter, recall that a clause is meant to represent a disjunction of its literals. Therefore, an interpretation I satisfies a clause $c = [l_1 \cdots l_n]$ iff I satisfies *some* l_j. Clearly, no such l_j exists when $n = 0$, so an empty clause is unsatisfiable.

DPLL can be viewed as a constraint-solving algorithm that must assign appropriate values to each of a number of variables, or else announce that the input constraint is not solvable. In this case the variables are the atoms in the given CNF input and the values are `true` and `false`, so a variable assignment is essentially an interpretation of the corresponding sentence. The assigned values must satisfy (solve) the input constraint. Many problems, ranging from solving systems of equations to unification and mathematical programming, can be formulated as constraint-solving problems. And there is a general methodology for solving such problems that can be described—at a high level of detail—as follows:

21 For the early history behind this algorithm, refer to Davis [26].

1. If the constraint is in a trivial form that directly represents a solution (positive or negative), then stop and give the relevant output.
2. Otherwise, choose a hitherto unassigned variable x and assign to it a value v.
3. Compute and propagate some local consequences of that assignment, thereby obtaining a (hopefully simpler) constraint.
4. Repeat.

DPLL (roughly) follows the same pattern, except that the propagation occurs at the beginning of the algorithm, so it computes either consequences of a variable assignment from the previous iteration or consequences of structural properties of the input. The process of propagating consequences is known as *boolean constraint propagation*, or BCP for short. In the case of DPLL, BCP consists of two transformations: *unit clause propagation* and *pure literal propagation*. We will describe both shortly. Using this terminology, the core algorithm for computing `(dpll L)` for a given list of clauses L can be described as follows:

- If the input list L is empty, return `true`.
- If the input list L contains an empty clause, return `false`.
- If BCP (boolean constraint propagation) is applicable to L, apply it to get a new list of clauses L' and then apply `dpll` recursively to L'.
- Otherwise, pick a currently unassigned literal l and return

  $$\texttt{(||\ (dpll\ (add\ [}l\texttt{]\ }L\texttt{))\ (dpll\ (add\ [}\overline{l}\texttt{]\ }L\texttt{)))},$$

  where $\overline{l}$ is the complement of l.

This version only returns `true` or `false`, but we will see that it is easy to instrument it so that it returns a satisfying interpretation when there is one.

We now turn to the two BCP transformations, starting with unit clause propagation. A *unit clause* is simply a clause with only one literal, such as `[A]` or `[(~ B)]`. Let us write l_u for the unique literal of any unit clause u. Thus, if u is the unit clause `[(~ B)]`, then $l_u =$ `(~ B)`.

Clearly, if our input list L contains a unit clause u then any interpretation that satisfies L *must* satisfy l_u. Accordingly, we can use u to simplify other clauses in L as follows:

1. If a clause c in L contains l_u then it must be satisfied by any interpretation that satisfies L, so we can remove c from L.
2. If a clause c contains $\overline{l_u}$ then we can remove $\overline{l_u}$ from c because, again, any interpretation I satisfying L must satisfy l_u, and therefore if I is to satisfy c, it must satisfy one of the other literals of c.

For example, suppose

$$L = [c_1\ c_2\ c_3\ c_4], \tag{4.16}$$

where

$$\begin{aligned} c_1 &= [\texttt{A}], \\ c_2 &= [\texttt{A (\~{} B) C}], \\ c_3 &= [\texttt{(\~{} C) (\~{} A) D}], \\ c_4 &= [\texttt{(\~{} D)}]; \end{aligned}$$

and let us focus on the unit clause $u = c_1 = [\texttt{A}]$. (There is one more unit clause here, `[(~ D)]`, but we can ignore that one for now.) For starters, we can remove c_2 from L because any interpretation of L must satisfy `A` and therefore also $c_2 = [\texttt{A (\~{} B) C}]$ (recall that a clause is a disjunction of its literals). In fact we can remove u itself (because it certainly contains l_u!), although we will remove unit clauses with care—we will store l_u in the interpretation that we will be incrementally building. And finally, we can remove $\overline{l_u} =$ `(~ A)` from c_3. Thus we are left with two clauses after this round of unit clause propagation:

`[(~ C) D]` and `[(~ D)]`.

There is a unit clause in this set of clauses as well, namely `[(~ D)]`, so we can apply one more round of unit propagation. This would then leave us with a single clause:

`[(~ C)]`,

which is itself a unit clause that could then be removed, at which point we would be left with the empty list of clauses, which is trivially satisfiable. Retrieving the unit literals that we removed along the way would then give us a satisfying interpretation, consisting of `A`, `(~ D)`, and `(~ C)`.

Note that this particular problem was solved purely with BCP, and in fact only with unit clause propagation. We didn't even get a chance to get to the hard part, which is the selection of some previously unassigned literal and the subsequent case analysis. Many easy problems are indeed solved just by BCP, which can be implemented very efficiently. Another example is given by the `if-chain` tautologies above. Consider again the tautology produced by `(if-chain 3)`, which is essentially the conditional which says that if `P1` and `(P1 ==> P2)` and `(P2 ==> P3)` all hold, then `P3` also holds (where, for simplicity, we write `P1` instead of `(P 1)`, etc.). To verify that this is a tautology using DPLL, we would try to verify that its negation is unsatisfiable. If we negate this conditional and convert to CNF, we end up with the following clauses:

`[[P1] [(~ P1) P2] [(~ P2) P3] [(~ P3)]]`.

A few repeated applications of unit propagation to this input will leave us with the clause list `[[]]`, a singleton consisting of the unsatisfiable empty clause.

Let L be a list of clauses and let l be a literal that appears in some of the clauses of L. We say that l is *pure* w.r.t. L iff the complement $\overline{l}$ does not appear anywhere in L. Thus, for instance, the clause list (4.16) has only one pure literal, `(~ B)`, because its complement `B` does not appear in any of the clauses of (4.16). All other literals of (4.16) are impure because they have *both* negative and positive occurrences in the given clause list. The *pure-literal transformation* removes from L all those clauses that contain pure literals $l_1, \ldots, l_m$. The resulting list L' is satisfiable iff L is. One direction of this claim is easy: If I is any interpretation that satisfies L, then it also satisfies L', since L' is a subset of L.[22] In the other direction, suppose that I is any interpretation satisfying the reduced set of clauses L'. Then we can easily extend I to a new interpretation that satisfies L by mapping each positive pure literal l_i to `true` and each negative pure literal l_j to `false`, for $i,j \in \{1, \ldots, m\}$. Since the new interpretation differs from I only in the pure literals $l_1, \ldots, l_m$, it follows that it satisfies the L' part of L. And it satisfies all the remaining clauses as well (the removed clauses containing pure literals), because it satisfies all the pure literals. For example, the pure-literal transformation entitles us to go from

```
[ [A B] [(~ A) C] [(~ C) B] ]
```

to

```
[ [(~ A) C] ]
```

by removing the two clauses containing the pure literal `B`. It is *not* the case here that any single interpretation that satisfies one set of clauses must also satisfy the other. For instance, the interpretation that assigns `A` and `B` to `false` and `C` to true satisfies the reduced set of clauses but not the original. The claim, rather, is that each set of clauses is satisfiable iff the other is, so that the transformation "preserves satisfiability."

We will implement `dpll` as a single call to an auxiliary procedure `dpll0` that will do all the hard work. `dpll0` will take two arguments, the input list of clauses L and a list of literals I representing an interpretation. Initially I will be empty; it will be incrementally extended. Both BPC transformations will also take the same two inputs as `dpll0`: the current list of clauses L and the current interpretation I. And they will both return a triple of the following form:

`[flag` L' I'`]`,

where *flag* is a bit indicating whether the transformation (unit clause or pure-literal propagation) was successful. If *flag* is `true`, then L' is the resulting list of clauses (obtained from L by applying the transformation) and I' is resulting interpretation (obtained from I by taking into account the effects of the transformation). With these signatures, we can write `dpll` and `dpll0` as follows:

22 We often speak as if L were a set, since it is in fact meant to represent one.

```
define (dpll clauses) := (dpll0 clauses [])

define (dpll0 clauses I) :=
  match clauses {

    [] =>  I # empty list of clauses, return I.

  | (_ where (member? [] clauses)) => 'unsat # empty clause indicates unsat.

  | _ => let {[success? clauses' I'] := (unit-propagation clauses I)}
          check
           {success? => (dpll0 clauses' I')
          | else => let {[success? clauses' I'] := (pure-literal clauses I)}
                     check
                       {success? => (dpll0 clauses' I')
                       | else => let {l := (choose-lit clauses)}
                                   match (dpll0 (add [l] clauses) I) {
                                     (some-list L) => L
                                    | _ => let {l' := (complement l)}
                                             (dpll0 (add [l'] clauses) I)
                                    }
                        }
           }
}
```

Apart from the two BCP procedures, `unit-propagation` and `pure-literal`, we are also missing `choose-lit`, the procedure for selecting a literal for the case analysis. There are many different heuristics for making that selection, although none of them can guarantee an always-optimal selection. Perhaps the simplest most effective heuristic is to choose the literal that appears in the greatest number of clauses, the rationale being that assigning a value to that literal will help to simplify the largest possible number of clauses. The following is an implementation of this heuristic:

```
define (choose-lit clauses) :=
  let {counts := (HashTable.table);
       process := lambda (c)
                    (map-proc
                      lambda (l)
                        let {count := try { (HashTable.lookup counts l)
                                          | 0 }}
                          (HashTable.add counts [l --> (1 plus count)])
                      c);
       _ := (map-proc process clauses);
       L := (HashTable.table->list counts)}
    letrec {loop := lambda (pairs lit max)
                      match pairs {
                        [] => lit
                      | (list-of [l count] more) =>
                          check {(count greater? max) =>
```

```
                                    (loop more l count)
                                | else => (loop more lit max)}
                        }}
      (loop L () 0)
```

On average, this runs in linear time in the number of clauses. We could sidestep the need to traverse all the clauses in order to make a selection by passing around a data structure containing additional information about the unassigned variables, which could let us make a selection more efficiently. However, we would then need to do some additional bookkeeping in order to update that structure every time we modify the clause list.

We continue with the implementation of unit literal propagation:

```
1  define (unit-propagate l clauses) :=
2    let {l' := (complement l)}
3      (map-select lambda (c)
4                    check {(member? l c) => ()
5                           | else => (filter c (unequal-to l'))}
6                  clauses
7                  (unequal-to ()))
8
9  define (unit-propagation clauses I) :=
10     let {f := lambda (c)
11                 match c {
12                   [l] => l
13                 | _ => ()
14                 }}
15       (find-some-element
16         clauses
17         (unequal-to ())
18         f
19         lambda (l)
20           let {clauses' := (unit-propagate l clauses)}
21             [true clauses' (l added-to I)]
22         lambda ()
23           [false clauses I])
```

We look for the first unit clause we can find in the `clauses` list. If we find one, then the success continuation on lines 19–21 takes the literal `l` of that clause and produces a new list of clauses `clauses'` by carrying out the actual unit propagation, through the auxiliary procedure `unit-propagate`. We then return the triple `[true clauses' (l added-to I)]`. If no unit clause is found, then the failure continuation on lines 22–23 is applied. The implementation of `(unit-propagate l clauses)` is straightforward: Every clause that contains `l` is removed, and the complement of `l` is removed from every other clause. As with `choose-lit`, we could simplify the discovery of unit literals (so that we don't have to traverse the entire clause list in the worst case, as we do now) in a number of ways. One way would be to always keep the input clauses partitioned into unit and nonunit clauses, which

we could do by representing `clauses` as a pair of clause lists (unit and nonunit). When removing literals from clauses, either via unit propagation or the pure-literal transformation, we would then need to make sure that new clauses end up in the right partition, but this is straightforward. The implementation of `pure-literal` is worked out in Exercise 4.34.

To test `dpll` on a regular sentence p, we first need to convert p to a list of clauses. Athena provides a primitive procedure `cnf-core` for that purpose. It is a binary procedure whose first argument is the sentence p to be converted and whose second argument is a meta-identifier specifying the format of the output. This argument has three possible values:

- `'dimacs-list`;
- `'sentence-list`; and
- `'sentence`.

If `'dimacs-list` is specified, then the result of the conversion will be a list of clauses, where each clause is a list of integers. These integers represent literals. A negative integer is a negated atom, whereas a nonnegative integer is a positive atom. An integer atom in the output corresponds either to one of the atoms in the input p, or else it is a *Tseitin variable* introduced during the conversion (refer to Exercise 4.34 for more details). In order for the caller to know which integers correspond to which atoms in the input p, `cnf-core` also includes in its output a hash table from integers to the said atoms. More specifically, the output of `cnf-core` is a map m that should be thought of as a record with several fields. One of the fields is `'result`, which is the actual list of clauses, and another field is `'atom-table`, which is the aforementioned hash table. The remaining fields provide statistics about the conversion and the output clauses:

- `'cnf-conversion-time`;
- `'total-var-num`;
- `'tseitin-var-num`;
- `'clause-num`;
- `'table-formatting-time`; and
- `'clause-formatting-time`.

The value of `'total-var-num` represents the total number of variables in the output clauses, while `'tseitin-var-num` gives the number of Tseitin variables introduced by the conversion. The sum of `'tseitin-var-num` and

```
(HashTable.size (m 'atom-table))
```

should be equal to `'total-var-num`, as any variable in the output is either a Tseitin variable or a variable corresponding to some atom in the input. The value of `'cnf-conversion-time` is the number of seconds spent on the CNF conversion. The other two time outputs give

the seconds spent on the other two tasks that are needed to prepare the output of `cnf-core`, namely formatting the table and the clauses. These numbers are typically smaller than the value of `'cnf-conversion-time`. The sum of these numbers should be the total time consumed by an invocation of `cnf-core`. An example:

```
define p := (A & B | C & D)

> define m := (cnf-core p 'dimacs-list)

Map m defined.

> m

Map:
|{
'result := [[7] [(- 3) 2] [(- 3) 1] [(- 6) 5] [(- 6) 4] [(- 7) 6 3]]
'atom-table :=
|[1 := B,
  2 := A,
  4 := D,
  5 := C
]|
'cnf-conversion-time := 0.0
'total-var-num := 7
'tseitin-var-num := 3
'clause-num := 6
'table-formatting-time := 0.0
'clause-formatting-time := 0.0
}|

> (m 'result)

List: [[7] [(- 3) 2] [(- 3) 1] [(- 6) 5] [(- 6) 4] [(- 7) 6 3]]

> (HashTable.lookup (m 'atom-table) 1)

Sentence: B
```

The `'dimacs-list` format is convenient for preparing text files in DIMACS format, which is the standard input format for SAT solvers.[23] The primitive Athena procedure `cnf` is defined as a wrapper call around `cnf-core` with `'dimacs-list` as its value:

```
define (cnf p) := (cnf-core p 'dimacs-list)
```

If `'sentence-list` is given as the second argument of `cnf-core`, then the `'result` field of the output map will be a list of clauses expressed as sentences, that is, as native Athena

23 See the information on `DIMACS-CNF` in the Wikipedia page on the *Boolean Satisfiability Problem*: `http://en.wikipedia.org/wiki/Boolean_satisfiability_problem`.

disjunctions or literals. A Tseitin variable such as x_2 will appear as `?X2:Boolean` in an Athena clause. For example:

```
define m := (cnf-core (A & B | C & D) 'sentence-list)

> (m 'result)

List: [
?X7:Boolean

(or (not ?X3:Boolean)
    A)

(or (not ?X3:Boolean)
    B)

(or (not ?X6:Boolean)
    C)

(or (not ?X6:Boolean)
    D)

(or (not ?X7:Boolean)
    ?X6:Boolean
    ?X3:Boolean)
]
```

Finally, if the second argument of `cnf-core` is `'sentence` then the `'result` field of the output map will be an Athena sentence in CNF (i.e., a conjunction of disjunctions of literals):

```
> ((cnf-core (A & B | C & D) 'sentence) 'result)

Sentence: (and ?X7:Boolean
               (or (not ?X3:Boolean)
                   A)
               (or (not ?X3:Boolean)
                   B)
               (or (not ?X6:Boolean)
                   C)
               (or (not ?X6:Boolean)
                   D)
               (or (not ?X7:Boolean)
                   ?X6:Boolean
                   ?X3:Boolean))
```

Note that `cnf-core` can handle arbitrarily complicated atoms, for example:

```
define p := (?x < ?y ==> ?x + 1 < ?y + 1)

> (cnf p)
```

```
Map:
|{
'result := [[2 (- 1)]]
'atom-table := |[

1 :=
(< ?x:Int ?y:Int)

2 :=
(< (+ ?x:Int 1)
   (+ ?y:Int 1))

]|
'cnf-conversion-time := 0.0
'total-var-num := 2
'tseitin-var-num := 0
'clause-num := 1
'table-formatting-time := 0.0
'clause-formatting-time := 0.0
}|
```

Even quantified sentences can serve as atoms:

```
define p := (forall ?x . ?x = ?x)

> (cnf (p & ~ p))

Map:
|{
'result := [[1] [(- 1)]]
'atom-table := |[1 :=
(forall ?x:'T3399
  (= ?x:'T3399 ?x:'T3399))
]|
'cnf-conversion-time := 0.0
'total-var-num := 1
'tseitin-var-num := 0
'clause-num := 2
'table-formatting-time := 0.0
'clause-formatting-time := 0.0
}|
```

We can now test `dpll` as follows:

```
define (test-dpll p) :=
  let {m := (cnf-core p 'sentence-list);
       clause-sentences := (m 'result);
       clauses := (map get-disjuncts clause-sentences)}
    (dpll clauses)

set-precedence test-dpll 10;;
```

```
> (test-dpll A ==> B)

List: [
(not A)
 B]

> (test-dpll ((A ==> B) & A & ~ B))

Term: 'unsat

> (test-dpll ~ if-chain 20)

Term: 'unsat
```

Even this rudimentary implementation of `dpll` is much more efficient than the brute-force approach of `taut?`. For example, it recognizes `(~ if-chain 400)` as unsatisfiable in a fraction a second, whereas the exhaustive search procedure would never finish given the number of interpretations it would need to generate and test (2^{400}). As we pointed out earlier, all instances of this particular problem are solvable by unit clause propagation alone, but `dpll` has decent performance on harder problems as well. Beyond the simple improvements to which we have already alluded (slightly better organized data structures to facilitate the retrieval of unit clauses and literal selection for case analysis), there are several techniques that can be deployed on top of this basic infrastructure and that typically result in dramatic performance improvements, such as *conflict learning* and *backjumping*. For further information and implementation details refer to *The Handbook of Propositional Satisfiability* [10].

Industrial-strength SAT solvers can be invoked in Athena through the primitive `sat-solve` procedure, which takes an arbitrary sentence p as input (or a list of sentences, see below), converts p to CNF as needed, prepares a DIMACS file, feeds that file to a SAT solver,[24] and then extracts the output of the SAT solver and formats it in a way that makes sense at the Athena level. The result of a call to `sat-solve` is a map containing various meta-identifier fields. The main output field is `'satisfiable`, which is `true` or `false` depending on whether the input p was found to be satisfiable. There is also an `'assignment` field. When `'satisfiable` is `true`, the value of the `'assignment` field is a hash table representing an interpretation of p, mapping atoms of p to `true` or `false`. When `'satisfiable` is `false`, the value of `'assignment` is the unit `()`. Other fields in the output map include:

- `'cnf-conversion-time`;
- `'sat-solving-time`;

24 Usually some (possibly modified) version of MiniSat.

- `'dimacs-prep-time`;
- `'total-var-num`;
- `'tseitin-var-num`; and
- `'clause-num`,

all of which should be self-explanatory in view of our foregoing discussion of `cnf-core`.

Here are some examples of `sat-solve` in action:

```
define p := (A ==> B)

> (sat-solve p)

Map:
|{
'cnf-conversion-time := 0.0
'total-var-num := 2
'tseitin-var-num := 0
'clause-num := 1
'satisfiable := true
'assignment := |[A := false, B := false]|
'sat-solving-time := 0.326
'dimacs-prep-time := 0.001
}|

> (sat-solve (p & ~ p))

Map:
|{
'cnf-conversion-time := 0.0
'total-var-num := 2
'tseitin-var-num := 0
'clause-num := 3
'satisfiable := false
'assignment := ()
'sat-solving-time := 0.288
'dimacs-prep-time := 0.001
}|
```

As with `cnf-core`, the input p can contain arbitrarily complicated atoms, including quantified sentences. For example:

```
define p := (forall ?x . ?x = ?x)

> (sat-solve (p & ~ p))

Map:
|{
'cnf-conversion-time := 0.0
'total-var-num := 1
```

```
'tseitin-var-num := 0
'clause-num := 2
'satisfiable := false
'assignment := ()
'sat-solving-time := 0.389
'dimacs-prep-time := 0.001
}|
```

Also, as mentioned earlier, the input to `sat-solve` can be a list of sentences L instead of just one single sentence. The output will then indicate whether all the elements of L are jointly satisfiable, and if so, the `'assignment` table will provide an appropriate interpretation:

```
> ((sat-solve [A (A ==> B) (~ A)]) 'satisfiable)

Term: false
```

The satisfiability problem is not just of theoretical interest. Because SAT is so expressive, many other difficult combinatorial problems have fairly straightforward reductions to it. Owing to the remarkable progress of SAT-solving technology in recent times, it is now feasible in some cases to reduce a tough problem to a SAT formulation and throw an off-the-shelf SAT solver at it instead of coding up a custom-made algorithm for the problem. Athena is particularly suitable for such problems because the reductions can be expressed fluidly and the integration with SAT solvers is seamless. In what follows we demonstrate this type of problem solving by reducing graph coloring—for an arbitrary number of colors—to SAT.

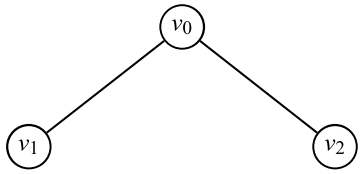
Graph coloring is a key problem in computer science that surfaces in many contexts under somewhat different guises. Mathematically, assuming we have k colors available $c_1, \ldots, c_k$, the problem asks whether it is possible to assign one of these colors to each vertex of a given undirected graph G in such a way that adjacent vertices receive distinct colors. If so, we say that G is *k-colorable*. If the answer is affirmative, we usually want the solution to produce such a coloring.

With sufficient imagination on what counts as a color and what counts as a vertex and an edge, it is possible to express many practical problems as instances of graph coloring, ranging from job scheduling and register assignment in compiler code generation to frequency allocation in radio communications and many more. For $k = 2$ colors the problem can be solved in polynomial time, as it reduces to the question of whether the graph G is bipartite. But for $k > 2$ the problem is NP-complete. For $k > 3$ the question always has a positive answer for *planar* graphs:[25] With 4 or more colors available it is possible to color *any* planar graph so that adjacent vertices are given distinct colors. That is the content of the famous four-color theorem. (Of course, in practice we are usually interested in discovering

25 A graph is planar if it can be drawn on the plane so that its edges do not intersect anywhere (other than their endpoints).

such a coloring, not simply knowing that one exists.) For $k = 3$, by contrast, there are many simple examples of planar graphs that are not k-colorable; we will give such an example shortly. The smallest k for which a graph is k-colorable is called its *chromatic number*. In some problems we are interested in a k-coloring for the smallest possible k, as the "colors" in those cases are resources that we want to minimize. The procedure we will write takes an arbitrary graph G and number of available colors k and produces a k-coloring of G, if one exists, or else reports that there is no such coloring.

The first question we need to answer is how to represent graphs. We will opt for two representations, one of which is intuitive and convenient for input purposes but not terribly efficient; and another "internal" representation that is much more efficient. It will be easy to convert from one to the other. The first representation encodes an undirected graph simply as a list of pairs of nodes, where a node is always identified by a nonnegative integer i. Thus, for instance, the following graph



(4.17)

is represented by the list `[[0 1] [0 2]]`. The "internal" representation of a graph G with N nodes $v_0, \ldots, v_{N-1}$ will be a pair consisting of the number N and a vector V of length N, where the i^{th} element of V contains a list of all and only the neighbors (adjacent vertices) of v_i. The following procedure takes a list of edges and produces a graph under this canonical representation. For simplicity, the number of nodes N is assumed to be 1 plus the maximum index of any node appearing in the given edges. The procedure could be easily modified to take N as an additional parameter, instead of computing it in this way.

```
define (make-graph edges) :=
  let {N := (max* (join (map first edges) (map second edges)));
       V := make-vector (1 plus N) [];
       _ := (map-proc lambda (e)
                        match e {
                          [i j] => let {N-i := vector-sub V i;
                                        N-j := vector-sub V j;
                                          _ := vector-set! V i (add j N-i)}
                                   vector-set! V j (add i N-j)
                        }
                      edges);
       _ := (map-proc lambda (i) vector-set! V i (dedup vector-sub V i)
                      (0 to N))}
   [(1 plus N) V]
```

Note that we add each edge in both directions and dedup all neighbor lists at the end.

```
define E := [[0 1] [0 2]]

> (make-graph E)

List: [3 |[1 2] [0] [0]|]
```

We use two simple datatypes to represent colors and nodes. Neither is necessary, as both colors and nodes are essentially integers, but they make the code more readable. We also introduce a binary predicate has to represent an assignment of a color to a node. Thus,

(node *i* has color *j*)

will have the obvious meaning.

```
datatype Color := (color Int)

datatype Node := (node Int)

declare has: [Node Color] -> Boolean
```

All we now have to do is write a procedure coloring-constraints that takes a graph G and the number K of available colors and produces a list of sentences, using has atoms, that captures the coloring constraints. We need three groups of constraints to that end. First, we must ensure that every node is assigned *at least* one color. Second, that every node is assigned *at most* one color. (Clearly, these two sets of constraints jointly imply that every node is assigned exactly one color.) And finally, for every node v_i we must ensure that if v_i is assigned a color c, then none of its neighbors are assigned c. We then join these three groups of constraints together and return the resulting list as the output of coloring-constraints. The code provides a good illustration of a situation in which polyadic conjunctions and disjunctions come handy:

```
define (coloring-constraints G K) :=
  match G {
   [N neighbors] =>
     let {all-nodes := (0 to N minus 1);
          all-colors := (1 to K);
          at-least-one-color :=
            (map lambda (i)
                   (or (map lambda (c)
                             (node i has color c)
                           all-colors))
                 all-nodes);
          at-most-one-color :=
            (map lambda (i)
                   (and (map lambda (c)
                              (if (node i has color c)
                                  (and (map lambda (c')
```

```
                                              (~ node i has color c')
                                             (list-remove c all-colors))))
                              all-colors))
                 all-nodes);
          distinct-colors :=
            (map lambda (i)
                 (and (map lambda (j)
                              (and (map lambda (c)
                                          (node i has color c ==>
                                             ~ node j has color c)
                                        all-colors))
                             vector-sub neighbors i))
                 all-nodes)}
      (join at-least-one-color at-most-one-color distinct-colors)
  }
```

Here are two of the constraints we get for graph (4.17) for $K = 2$; the first expressing that the first node `(node 0)` must be assigned at least one color, and the second one expressing the distinctness requirement for `(node 0)`:

```
(or (has (node 0)
         (color 1))
    (has (node 0)
         (color 2)))

(and (and (if (has (node 0)
                   (color 1))
              (not (has (node 1)
                        (color 1))))
          (if (has (node 0)
                   (color 2))
              (not (has (node 1)
                        (color 2)))))
     (and (if (has (node 0)
                   (color 1))
              (not (has (node 2)
                        (color 1))))
          (if (has (node 0)
                   (color 2))
              (not (has (node 2)
                        (color 2))))))
```

Note that this implementation duplicates distinctness constraints for adjacent vertices. For instance, if v_2 and v_5 are adjacent and we state that if v_2 is red then v_5 should not be red, we will later also state that if v_5 is red then v_2 should not be red, which will be superfluous at that point. Generally speaking, in industrial applications of SAT solving we try to minimize

the number of generated constraints,[26] and so in a case like this we would not generate the redundant constraints. This can be easily done here by keeping a hash table of all generated distinctness constraints involving an edge `[i j]`. We leave that as an exercise (Exercise 4.37).

We now write a top-level `graph-coloring` procedure that takes a list of edges and the number of available colors K and either produces a K-coloring of the graph G represented by the given edges or else outputs the unit value `()`, indicating that no such coloring exists. A third parameter is a map M assigning strings to color codes, that can be used to print the output coloring in a more readable way. For instance, we can agree that colors 1, 2, and 3 are "red," "green," and "blue," respectively. If M does not assign a string to a given color c_i, then the color is printed simply as `(color i)`, so we can pass the empty mapping `|{}|` as the value of M if we don't care to specify any color names.

```
define (show-results L color-names) :=
  let {_ :=  (map-proc lambda (p)
               match p {
                  [((node i) has (color c)) true] =>
                    let {name := try {(color-names c)
                                      | (val->string color c)}}
                      (print "\nNode" i "-->" name)
               | _ => ()
               }
             L)}
    (print "\n")


define (graph-coloring edges K m) :=
  let {G := (make-graph edges);
       constraints :=  (coloring-constraints G K);
       res := (sat-solve constraints)}
   match (res 'assignment) {
     (some-table ht) => (show-results (rev HashTable.table->list ht) m)
    | r => (print K "colors are not enough for this graph.")
   }

define color-names :=
  |{1 := "red", 2 := "blue", 3 := "green", 4 := "yellow"}|
```

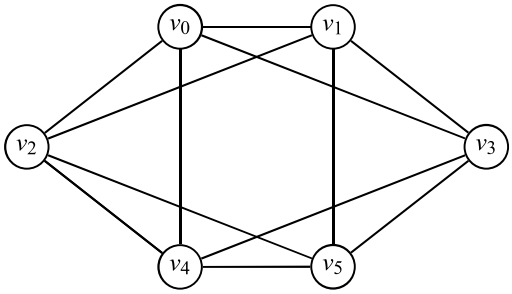
Let's test the algorithm with the simple graph (4.17):

```
> (graph-coloring [[0 1] [0 2]] 2 color-names)

Node 0 --> blue
Node 1 --> red
```

26 However, the relationship between the total number of constraints (or CNF clauses, ultimately) and the search complexity of the resulting SAT problem is not always clear. Sometimes adding constraints can actually speed up the search.

```
Node 2 --> red

Unit: ()
```

The following graph makes for a more interesting example:



(4.18)

It should be clear that it cannot be colored with 2 colors alone. Can it be colored with 3? Let's define it first as a list of edges (note that we don't need to repeat symmetric edges, for example, we do not list `[1 0]` given that we have listed `[0 1]`):

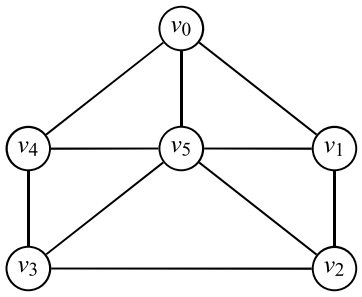
```
define g := [[0 1] [0 3] [0 4] [0 2]
             [1 2] [1 3] [1 5]
             [2 4] [2 5]
             [3 4] [3 5]
             [4 5]]
```

The answer, as we see, is affirmative:

```
> (graph-coloring g 3 color-names)

Node 0 --> green
Node 1 --> blue
Node 2 --> red
Node 3 --> red
Node 4 --> blue
Node 5 --> green
```

Figure 4.1 provides a visual representation of this coloring (writing *G* for green, etc.).

As a negative example, and perhaps somewhat surprisingly, it can be shown that the following graph has no 3-coloring:



Let's verify this:

```
define g := [[0 4] [0 5] [0 1]
             [1 2] [1 5]
             [2 5] [2 3]
             [3 5] [3 4]
             [4 5]]

> (graph-coloring g 3 color-names)

3 colors are not enough for this graph.
```

In all of these examples answers were returned instantaneously. For more extensive testing, we can write a procedure that generates random undirected graphs. One way to do that is through the *Gilbert random graph model* $G(N,p)$, determined by the number of vertices n and a real number $p \in (0,1)$, where the existence of an edge between any two vertices in the graph has independent probability p. A procedure for generating random Gilbert graphs can be coded in Athena as follows:

```
1 define (make-random-graph N p) :=
2  let {all-nodes := (0 to N minus 1);
3       all-edges := (filter-out (all-nodes X all-nodes)
4                                lambda (p) (first p equal? second p))}
5    (filter all-edges lambda (_) (bernoulli-trial p))
```

where a Bernoulli trial with probability of success p is defined as follows:

```
define (bernoulli-trial p)  :=
  let {log := (length tail tail val->string p);
       limit := (10 raised-to log);
       r := (random-int limit)}
    (r leq? p times limit)
```

The binary procedure `X` (line 3 in the listing of `make-random-graph`) builds the Cartesian product of two lists. For example:

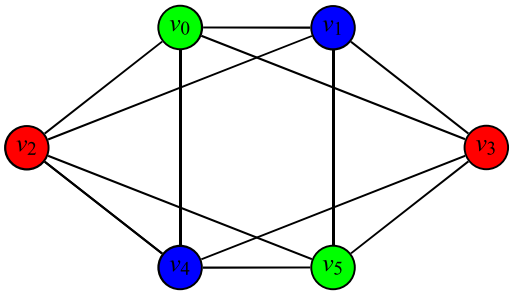


Figure 4.1
A coloring for graph (4.18) discovered by SAT solving.

```
> ([1 2 3] X ['a 'b])

List: [[1 'a] [1 'b] [2 'a] [2 'b] [3 'a] [3 'b]]
```

We can now test our code as follows:

```
define (test-gc N p K) :=
  (graph-coloring (make-random-graph N p) K |{}|)
```

Experimentation will show that, depending on how connected the graph is, the SAT solver can quickly find colorings even for hundreds or thousands of nodes, and for hundreds of thousands of constraints.

A drawback of the Gilbert model is that it examines every possible edge $e = v_i, v_j$ to decide whether to keep e. Therefore its run time is quadratic in the number of vertices, which can be nontrivial even for a few thousand vertices. An alternative technique for generating random graphs over a given number of vertices is to take as input the required number of edges, and then keep generating random pairs of vertices until we have the desired total number of (distinct) edges. We leave the implementation of that approach as an exercise.

Exercise 4.1 (Sentence Polarities): Let p be the sentence `(A & ~ B)`. We say that the occurrence of `A` in p is *positive*, or that it has a positive *polarity*, whereas the occurrence of `B` is *negative*, or that it has a negative polarity. Intuitively, this is intended to reflect the fact that `B` is embedded inside p within an odd number of negation signs,[27] while `A` is within an even—in this case zero—number of negations. Likewise, the occurrence of `B` in

```
(~ ~ B | ~ C)
```

27 Meaning that if we traverse the abstract syntax tree of p from the root until `B`, we will encounter an odd number of negations along the way.

is positive, as is the occurrence of `(~ C)`. However, the occurrence of `C` is negative, as is the occurrence of `(~ B)`. Of course, for any p, the unique occurrence of p in p itself is positive.

It is possible for a sentence q to have both positive and negative occurrences inside a given p. For instance, `(A | B)` has two occurrences in

$$(\underline{(A \mid B)} \mid (C \,\&\, \sim \overline{(A \mid B)})),$$

the first of which (the underlined occurrence) is positive, while the second (overlined) occurrence is negative.

When it comes to conditionals and biconditionals, polarity it not just a matter of traversing an abstract syntax tree and counting negations. Consider first a conditional $(p \texttt{ ==> } q)$. Because this is essentially equivalent to

$$(\sim p \mid q),$$

we regard the occurrence of p in the conditional to be negative, and the occurrence of q to be positive. By the same token, since a biconditional

$$(p \texttt{ <==> } q) \tag{4.19}$$

is equivalent to the conjunction of the two conditionals $(p \texttt{ ==> } q)$ and $(q \texttt{ ==> } p)$, we regard the occurrence of p in (4.19) as *both* positive and negative, and likewise for the occurrence of q. Similarly, every occurrence of a subsentence of p in (4.19) is both positive and negative, and likewise for every occurrence of a subsentence of q in (4.19). We will use the term `'pn` to denote a polarity that is both positive and negative—or "positive-negative." The terms `'p` and `'n` will denote positive and negative polarities, respectively.

We can make the above discussion more precise by defining a binary Athena procedure `polarity` that takes two arbitrary sentences p and q and returns `'p`, `'n`, or `'pn` depending on whether p has a positive, negative, or positive-negative occurrence in q, respectively. However, this simple interface would not be able to handle cases in which (a) p has various occurrences of different types in q (e.g., one positive occurrence and one negative); and (b) p has no occurrences in q. A better specification is this: Return *a list* of terms of the form `'p`, `'n`, or `'pn`, one for each occurrence of p in q. If p does not occur in q, the empty list should be returned. We thus name the procedure in the plural, `polarities`, and define it as follows:

```
define (flip pol) :=
  match pol {
    'p => 'n
  | 'n => 'p
  | 'pn => 'pn}

define (polarities p q) :=
  match q {
    (val-of p) => ['p]
  | (~ q1) => (map flip (polarities p q1))
```

```
  | (q1 ==> q2) => (join (map flip (polarities p q1))
                        (polarities p q2))
  | (q1 <==> q2) => (map lambda (_) 'pn
                        (join (polarities p q1) (polarities p q2)))
  | ((some-sent-con _) (some-list args)) =>
       (flatten (map lambda (q) (polarities p q)
                     args))
  | _ => []
  }
```

The body of the procedure is a **match** expression that starts by checking whether p and q are identical. If so, then `['p]` is returned, as p has exactly one occurrence in q, and it is a positive one. Otherwise we check to see if q is a negation of some q_1. In that case we recursively obtain the polarities of p in q_1 and flip them (positives become negatives and vice versa, whereas positive-negative occurrences remain unchanged) to account for the outer negation. If q is a conditional, we recursively obtain the polarities of p in the antecedent and flip them (since we view the antecedent as implicitly negated), and then we concatenate those to the polarities of p in the consequent. If q is a biconditional `(`q_1 `<==>` q_2`)`, we recursively obtain the polarities of p in q_1 and in q_2, and we turn all of them into positive-negative polarities. Next, if q is a conjunction or disjunction of the form

$$(\oplus\ q_1 \cdots q_n),$$

for $\oplus \in \{\mathsf{and}, \mathsf{or}\}$, we recursively compute the polarities of p in each q_i and then join them together.[28] Finally, if none of these conditions obtains then q must be an atom distinct from p, in which case p has no occurrences in q and the empty list is returned. Here are some examples of this procedure in action:

```
> (polarities A (A & (B | C)))

List: ['p]

> (polarities A (A ==> A | B))

List: ['n 'p]

> (polarities (~ A) (A ==> ~ B))

List: []

> (polarities (B & C)
              (~ C ==> (B & C <==> D | E)))

List: ['pn]
```

28 Recall the definition of `(flatten` L`)` as `(foldl join [] `L`)`; see Section 2.6.

```
> (polarities B
             (~ B ==> (B & C <==> ~ B | E)))

List: ['p 'pn 'pn]
```

(a) Note that `(polarities` p q`)` also gives us, indirectly, the *number* of occurrences of p in q; this is simply the length of the returned list. Modify the implementation so that every item in the output list consists of a *pair* (a two-element list)

$$[\textit{position polarity}]$$

comprising the *position* of the relevant occurrence as well as its *polarity*.[29] For instance,

```
(polarities A (A & B | A & C))
```

should return `[[[1 1] 'p] [[2 1] 'p]]`, indicating that there is one positive occurrence of `A` at position `[1 1]` and another one at `[2 1]`.

(b) Implement a binary procedure `polarities*` that takes a sentence p and a list of sentences L and returns a list of the form

$$[[q_1\ L_1]\ \cdots\ [q_k\ L_k]],$$

$k \geq 0$, where $q_1, \ldots, q_k$ are all and only those members of L that contain at least one occurrence of p,[30] and where each L_i is the result of applying the preceding version of `polarities` to p and q_i. That is, L_i is a list of all the positions and polarities of p inside q_i.

(c) Implement two ternary methods `M+` and `M-` that take as inputs (i) a premise of the form $(p$ `==>` $p')$; (ii) a sentence q; and (iii) a position u (a list of positive integers) such that q contains an occurrence of p at u that is strictly positive or strictly negative.[31] The methods should behave as follows:

- If the occurrence of p in q at u is positive, then `M+` should derive $(q$ `==>` $q')$, where q' is the sentence obtained from q by replacing the said occurrence of p by p'.
- If the occurrence of p in q at u is negative, then `M-` should derive $(q'$ `==>` $q)$, where q' is the sentence obtained from q by replacing that occurrence of p by p'.

`M+` should fail if the occurrence of p at u is not strictly positive, and `M-` should fail if the occurrence of p at u is not strictly negative. Both methods should fail if p does not occur

29 Recall from Section 3.3 that the position of an occurrence of p in q is a list of integers $[i_1 \cdots i_n]$ representing the path that must be traversed to get from the root of q to the said occurrence of p.

30 The order in which the various items $[q_i\ L_i]$ are listed is immaterial, but there should be no duplicates in the list.

31 Meaning that the occurrence is not `'pn`.

in q at u, or if the inputs are not of the specified forms. (Note: You will want to read Section 4.10 before tackling this part of the problem, as these two methods will have to be mutually recursive.) □

4.14 Proof heuristics for sentential logic

If we analyze the process that we went through in constructing the proof of Section 4.8 and try to depict it graphically, the result might look like the tree shown in Figure 4.2,[32] which we call a *goal tree*. Every node in such a tree contains a goal, which in the present context is a specification for a proof, or a *proof spec* for short. A proof spec is of the form "Derive p from β" where β is some assumption base, that is, a finite set of available assumptions, and p is the desired conclusion that must be derived from β. We use the letter τ as a variable ranging over proof specs (goals). By convention, we enclose every proof spec inside a box. Occasionally we also refer to the target conclusion p as a "goal." It will always be clear from the context whether we are using "goal" in reference to a target sentence within a proof spec or to a proof spec itself, as a whole.

The children of a proof spec τ, call them $\tau_1, \ldots, \tau_n$, $n \geq 0$, are obtained by applying a *tactic* to τ. In particular, applying a tactic to τ produces two things:

1. A partial deduction D—partial in that it may contain $n \geq 0$ applications of **force**. This partial deduction D is capable of deriving the target sentence p from the corresponding assumption base β, provided that the various gaps in it have been filled, that is, provided that the n applications of **force** have been successfully replaced by proper deductions.
2. A list of proof specs $\tau_1, \ldots, \tau_n$ that capture the proof obligations corresponding to the n applications of **force** inside D. We refer to these new proof specs as the *subgoals* generated by the tactic.

We then need to tackle these new goals $\tau_1, \ldots, \tau_n$ by applying new tactics to each τ_i, thereby expanding the goal tree to deeper levels. The process ends when the deductions at the leaves of the tree contain no occurrences of **force**, which typically happens when the deductions are single applications of `claim`. We visually mark such leaves by placing the sign $\surd$ right next to the corresponding deductions. When every branch of the goal tree has terminated at such a leaf, the process is complete. We can then scan the tree from the bottom up and put together complete definitions for the various partial deductions at every internal node, by incrementally eliminating applications of **force**, culminating with the top-level deduction at the root of the tree. Don't worry if not all of these details make full sense at this point; the ideas will start to sink in as we go along.

32 Tree structure here is indicated by indentation, which is one of several visual devices for drawing trees; see Knuth's discussion on p. 309 of his classic first volume [59].