In this section, we describe linear time algorithms for two syntactic restrictions of SAT. The algorithms are based on an efficient simplification process, called unit propagation, which also lies at the core of most high-performance SAT solvers.

**Terms and Conventions** In this section \( \Gamma \) denotes a set of clauses, with \( n \) distinct atoms, \( m \) clauses, and length (number of literals) \( l \), and \( \alpha \) is a (possibly partial) truth assignment for the atoms of \( \Gamma \). We will often write CNF formulas as sets of clauses, sometimes leaving out the braces, so that \((P \lor Q) \land (R \lor S)\) becomes \((P \lor Q), (R \lor S)\).

## 1 Unit Propagation

Unit propagation is an important simplification operation for CNF formulas based on the observation that, if \( \Gamma \) contains a unit clause \((P)\), then every satisfying assignment for \( \Gamma \) must make \( P \) true. If \( \Gamma \) also contains a clause \((\neg P \lor Q)\), we may now observe that it is “effectively unit”, because the only way to satisfy it is to set \( Q \) true. Unit Propagation (UP) is the name for the process of following this line of implications.

**Definition 1.** The “Unit Clause Rule” (UCR) is the following simplification rule for CNF formulas.

**UCR:** If \( \Gamma \) contains a unit clause \((L)\), delete every occurrence of \( L \) from clauses of \( \Gamma \), and delete every clause containing \( L \) from \( \Gamma \).

We denote by \( \text{UP}(\Gamma) \) the result of repeated application of UCR until there are no unit clauses in the resulting formula. That is, it is a fixpoint of UCR.
The fixpoint of UCR is unique, provided it does not contain (). If a fixpoint of UCR contains (), it need not be unique, but every fixpoint will contain () .

**Example 1.** Let \( \Gamma \) be
\[
(P), (P \lor Q \lor \neg R), (\neg P \lor R \lor U), (\neg P \lor \neg Q), (\neg Q \lor \neg P \lor R), (R \lor \neg U \lor V), (\neg P \lor Q \lor V)
\]
Applying UCR with \( (P) \), we obtain
\[
(R \lor U), (\neg Q), (\neg Q \lor R), (R \lor \neg U \lor V), (Q \lor V)
\]
Which produces a new unit clause \( (\neg Q) \). Now apply UCR with \( (\neg Q) \), and obtain
\[
(R \lor U), (R \lor \neg U \lor V), (V)
\]
Finally, we apply UCR with the new unit clause \( (V) \), obtaining
\[
(R \lor U).
\]

**Exercise 1.** Construct \( \text{UP}(\Gamma) \), where \( \Gamma \) is
\[
\]

**Proposition 1.** For any CNF formula \( \Gamma \):
1. \( \Gamma \) is satisfiable iff \( \text{UP}(\Gamma) \) is;
2. if \( () \in \text{UP}(\Gamma) \) then \( \Gamma \) is unsatisfiable.

In the next two sections, we show how to solve two important special cases of SAT efficiently using UP. In later notes we will see how efficient UP is a central component in industrial-strength SAT solvers. Naive computation of \( \text{UP}(\Gamma) \) takes quadratic time, as it involves repeated scanning of the entire formula to find all occurrences of a literal. However, with suitable data structures it can be computed in linear time. Later in this section of notes we will describe a method for doing this that is important in practice.

**Proposition 2.** \( \text{UP}(\Gamma) \) can be computed in time \( O(l) \).

We show how to do so in Section 4.

## 2 Horn-SAT

**Definition 2.** A clause is Horn if it contains at most one positive literal. Clause set \( \Gamma \) is Horn if each of its clauses is Horn. Horn-SAT is the problem of deciding satisfiability of sets of Horn clauses (a.k.a. Horn formulas).
We may think of the size of a truth assignment as being the number of atoms it makes true. Then every Horn formula has a unique minimal model, and we can find this in linear time.

**Proposition 3.** If a Horn formula has no positive unit clauses, then it is satisfiable.

**Exercise 2.** Prove Proposition 3. (Hint: This is a special case of the property that an arbitrary CNF formula with no positive clause is satisfiable.)

In fact, the following process shows that we can compute the minimal model of a Horn formula using UP. First, partition \( \Gamma \) into two sets: \( \Gamma^- \) containing only the negative clauses of \( \Gamma \) (i.e., those clauses containing no positive literal); and \( \Gamma^+ \), containing only those clauses which have a positive literal. Now compute UP(\( \Gamma^+ \)), and construct the partial truth assignment \( \tau \) of atoms set true during the computation. If \( \tau \) makes a clause of \( \Gamma^- \) false, then \( \Gamma \) is unsatisfiable. Otherwise, \( \Gamma \) is satisfiable, and the total assignment obtained from \( \tau \) by setting all remaining atoms to false is its minimal model.

To see why this it the case, first observe that the only unit clauses which appear during computation of UP(\( \Gamma^+ \)) are positive. We can show this by induction on the number of unit clauses generated. The base case is zero: the initial formula has only positive unit clauses, by construction. If re-write each non-unit clause \( (\neg P_1 \lor \neg P_2 \lor \ldots \lor \neg P_k \lor P) \) of \( \Gamma^+ \) as the equivalent formula \( (P_1 \land P_2 \land \ldots \land P_k \rightarrow P) \), then it is easy to see that, in each step where a new unit clause is obtained, it must be positive (otherwise there was a negative unit clause which made \( P \) false).

Thus, we see that \( \tau \) sets some atoms true, and (by Proposition 1) those atoms are true in every satisfying assignment for \( \Gamma \). If clause of \( \Gamma^- \) is made false by the assignments in \( \tau \), then \( \Gamma \) is unsatisfiable. Otherwise, all remaining atoms can be set false, and this will be sufficient to satisfy all clauses of \( \Gamma^- \).

**Exercise 3.** Compute the minimal model of

\[
(P \lor \overline{Q} \lor \overline{R}), (Q \lor \overline{S}), (R), (\overline{W} \lor S), (W), (\overline{W} \lor \overline{X} \lor P)
\]

This leads us to the following conclusion.

**Proposition 4.** Horn formula \( \Gamma \) is unsatisfiable if and only if UP(\( \Gamma \)) contains the empty clause.

**Corollary 1.** Horn-SAT can be solved in time \( O(1) \).
3 2-SAT

Definition 3. 2-SAT is the problem of deciding satisfiability of sets of clauses with at most two literals per clause (a.k.a. 2-CNF formulas).

There are two well-known linear-time algorithms for 2-SAT, one based on unit propagation, and one based on finding cycles in a graph associated with the formula. We focus on the former, which is more practical and is closely related to other algorithms we use. In Section 3.1 we give a brief sketch of the graph based method.

The UP-based algorithm is based upon the following observation. Suppose \( \Gamma \) is a 2-CNF and we apply UCR for some unit clause \( \{L\} \). Then for each clause \( C \) in \( \Gamma \) we have exactly one of:

1. \( C \) contains \( L \), so is now satisfied;
2. \( C \) contains \( \bar{L} \), so becomes a unit clause;
3. \( C \) is “untouched”.

It follows that UP(\( \Gamma \)) either contains the empty clause, or contains only clauses of size exactly two, none of which contain atoms which were given values during unit propagation.

For any literal \( L \) which occurs in \( \Gamma \), define UP(\( \Gamma, L \)) to be UP(\( \Gamma \cup \{L\} \)).

Proposition 5. If \( \Gamma \) is a 2-CNF in which \( L \) occurs, then exactly one of the following is true:

1. UP(\( \Gamma, L \)) contains the empty clause, in which case \( \Gamma \models \bar{L} \);
2. UP(\( \Gamma, L \)) is a proper subset of \( \Gamma \), and is satisfiable if and only if \( \Gamma \) is.

A simple polytime algorithm for 2-SAT, based on this proposition, is given as Algorithm 1. Assuming linear time unit propagation, the algorithm runs in time O(\( nl \)). Example 2 demonstrates that this upper bound is tight.

Example 2. Let \( \Gamma \) be the following conjunction of clauses:

\[
(\overline{P_1} \lor P_2) \land (\overline{P_2} \lor P_3) \land \ldots \land (\overline{P_{n-1}} \lor P_n) \\
\land (P_n \lor Q) \land (P_n \lor Q) \\
(P_1 \lor R_1) \land (P_2 \lor R_2) \land \ldots \land (P_n \lor R_n)
\]

(1)

\( \Gamma \) has \( 2n \) atoms, and \( 2n + 1 \) clauses. Consider a run of Algorithm 1 with input \( \Gamma \) in which the first literal chosen in line 3 is \( P_1 \), the second is \( P_2 \), etc. When \( P_0 \) is chosen, unit propagation sets all atoms \( P_i \), for \( i > 1 \), true, and then constructs the empty clause from \( (P_n \lor Q) \land (P_n \lor \bar{Q}) \). When \( P_1 \) is set in line 6, the clause \( (P_1 \lor R) \) is satisfied, and nothing else happens. The pattern is
Algorithm 1 Unit propagation based algorithm for 2-SAT

1: procedure UP-2SAT(Γ) // Γ is a set of binary clauses
2: while Γ is not empty do
3:   L ← a literal from Γ
4:   ∆ ← UP(Γ,L)
5:   if () ∈ ∆ then // we know Γ |= L
6:     ∆ ← UP(Γ,L)
7:     if () ∈ ∆ then // we know Γ |= L and Γ |= L
8:       return “UNSAT”
9:   end if
10: end if
11: Γ ← ∆
12: end while
13: return “SAT”
14: end procedure

similar after choosing P₂, P₃, etc. With linear time unit propagation, this algorithm has worst-case time Θ(nl), which can be quadratic in the input size.

To obtain a linear time algorithm, we must ensure that, whenever unit propagation does more than a very small amount of work, that work is reflected in a reduction in the size of the formula. We can arrange for this by performing the propagation processes for L and L in parallel, as shown in Algorithm 2. For purposes of describing this algorithm, we define “one step of unit propagation” to be “touching” (eliminating either by identifying as satisfied or identifying as a new unit) one clause. This algorithm runs in time O(I).

Exercise 4. It is well-known that 2-Col can be solved in polynomial time, and in fact linear time. Consider the representations of K-colouring as CNF given in the notes on CNF, for the special case when K = 2. What can we say about the CNF representations of 2-Col?

3.1 The Implication Graph Method

Here, we give a brief sketch of the other, and perhaps most common, method of showing 2-SAT is in linear time.

Definition 4. The implication graph IGΓ of a 2-CNF formula Γ is the directed graph IGΓ = ⟨V, A⟩ with vertices V = {P, ¬P | P is an atom occurring in Γ} and arc set A =
Algorithm 2 Linear-time Unit propagation based algorithm for 2-SAT

1: procedure Linear-2SAT(Γ) // Γ is a set of binary clauses
2: while Γ is not empty do
3:     L ← a literal from Γ
4:     Perform computation of UP(Γ,L) and UP(Γ,L) step-for-step in parallel;
5:     When the first of these processes terminates, halt the other also;
6:     if UP(Γ,L) halted first then
7:         let L denote L.
8:     end if
9:     if () ∉ UP(Γ,L) then
10:         Γ ← UP(Γ,L)
11:     else
12:         Γ ← UP(Γ,L)
13:         if () ∈ Γ then
14:             return “UNSAT”
15:         end if
16:     end if
17: end while
18: return “SAT”
19: end procedure

{((L,L′),(L,L′)) | (L ∨ L′) ∈ Γ}. If Γ contains a unit clause (L), we treat it as the logically equivalent 2-clause (L ∨ L).

The intuition is that the arcs of the graph capture the implications represented by each clause. A clause (L,L′) in Γ requires that, if L is false then L′ is true, and if L′ is false, then L is true.

Proposition 6. 2-CNF Γ is unsatisfiable off, for some atom P, GΓ has a directed cycle containing both P and P.

To check satisfiability of Γ, we generate IGr, find its strongly connected components, and then check if any strong component contains both an atom and its negation. Strongly connected components can be constructed in linear time.
Performing unit propagation in linear time requires using an appropriate data structure. The method described here is sufficient for showing the 2-SAT and Horn-SAT algorithms given above are linear time, and for effective implementations of these algorithms. More importantly, fast UP lies at the heart of modern industrial SAT solvers (other related software), and this is the standard method used in such solvers.

It is useful to be clear about what we need to do. Typically, we do not need to explicitly construct the formula \( \text{UP}(\Gamma,L) \). For example, in deciding Horn-SAT, we first want to know if performing UP generates the empty clause. If so, we don’t need to know anything else about \( \text{UP}(\Gamma,\text{lit}) \). To determine this, it is sufficient to be able to identify what new unit clauses arise (or if the empty clause arises), and ignore almost everything else. Rather than thinking of applying UCR literally, we imagine a process of constructing a partial truth assignment which assigns true to exactly the literals which appear in unit clauses. A clause is “effectively a unit clause” if all of its literals but one have been set false.

Now, suppose we have just detected the unit clause \((L)\). Our goal is to determine if setting \(L\) true will generate a new unit clause. We don’t care about clauses that don’t mention \(L\) or \(\overline{L}\). We also don’t care about clauses containing \(L\), as they will be satisfied. Now, consider a clause \(C\) containing \(L\), but also many other literals. “Setting \(L\) true” effectively makes \(C\) shorter by one, but \(C\) cannot become a unit clause if there are other literals which have not been made false. We want to avoid repeatedly visiting and scanning long clauses, except if they are in danger of having turned into unit clauses. To do this, we select two literals in each clause \(C\) to “watch”. As long as both watched literals of \(C\) remain non-false (i.e., they are either unassigned or assigned true), we can ignore \(C\). The only time we need to visit \(C\) is when one of its watched literals has been set false, at which point it may have been made effectively a unit clause, or even empty. The data TWL data structure maintains, for each literal \(L\), a list of watched occurrences of \(L\) in clauses of \(\Gamma\). This is illustrated in Figure 1.

The algorithm for performing UP with the TWL data structure is given in Algorithm 3. To ensure the algorithm executes in linear time, we must be careful about how we scan clauses when visiting them. To see why, consider formulas containing a clause \(C = (L_1,L_2,\ldots,L_k)\), where \(k\) is some constant fraction of \(l\) (the total number of literals in the formula). Suppose that we begin with literals \(L_1\) and \(L_k\) watched, that the order of assignments made by the algorithm is \(L_1,L_2,\ldots\), and that each time we visit \(C\) we do a sequential scan from left to right looking for a new literal to watch. It is clear that we spend \(\Omega(l)\) time just scanning the clause \(C\). However, if we use some reasonable strategy
to ensure we never inspect a literal of $C$ more than once (or some small constant number of times) during an entire run of the algorithm, then we can be sure the algorithm runs in $O(l)$ time. A simple way to do this is to maintain an index into $C$ which we always increase when doing a scan, so we inspect each literal only once. We also need to arrange $O(1)$ access to the second watched literal, which can be done in several ways.

### 4.1 TWL UP and 2-SAT

If the formula is 2-CNF, the TWL algorithm works as advertised, but is also overkill. Since each clause contains 2 literals, both of them are watched. Thus, the watch list for $L$ is essentially a list of all occurrences of $L$ in the formula. When we set $L$ false, all clauses containing $L$ become unit. Thus, it is sufficient to keep the list of these literals, and forget about the middle case of the if-elseif-else statement in the body of Algorithm 3.

---

Figure 1: The TWL data structure. The clause list appears across the top; the list of watch lists down the left side. Each clause has two watched literals.
Algorithm 3 Linear-Time Unit Propagation using Two Watched Literals

1: procedure UP-TWL($\Gamma$) // $\Gamma$ is a set of clauses
2: Q ← empty queue of literals;
3: $\alpha$ ← an empty partial truth assignment
4: for each unit clause $(L)$ in $\Gamma$ do
5: set $\alpha(L)$ true, and enqueue $L$ on $Q$
6: end for
7: Construct the TWL data structure for all non-unit clauses of $\Gamma$.
8: while Q is not empty do
9: $L$ ← a literal dequeued from $Q$
10: for each clause $C$ in the list of watched occurrences of $L$ do
11: // We were watching $L$ in $C$, but $L$ has been made false
12: $H$ ← the other watched literal of $C$
13: if there is a non-false literal $J$ in $C \setminus \{H, L\}$ then
14: add $C$ to the watch list for $J$
15: else if $\alpha(H) = false$ then // $C$ is effectively an empty clause
16: return “$(\) \in UP(\Gamma)$”
17: else // $C$ is effectively unit,
18: set $\alpha(H) = true$ and enqueue $H$ to $Q$
19: end if
20: end for
21: end while
22: return “$(\) \not\in UP(\Gamma)$”
23: end procedure