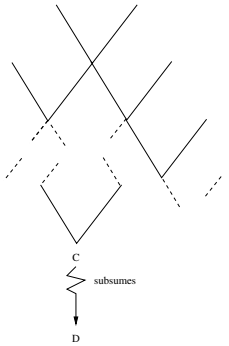
In general, it should be easy to see that if C and D are clauses such that $C \subseteq D$, then $C \models D$. In fact, for propositional logic, it is also the case that if $C \models D$ then C subsumes D (we see why this is so shortly).

The notion of subsumption acts as the basis for an important result linking resolution and logical implication, called the *subsumption theorem*:

Theorem 9 *If Σ is a set of clauses and D is a clause. Then $\Sigma \models D$ if and only if D is a tautology or there is a clause C such that there is a derivation of C from Σ using resolution ($\Sigma \vdash_R C$) and C subsumes D.*

By "derivation of a clause C" here, we mean the same as on page 30, that is, there is a sequence of clauses $R_1, \ldots, R_k = C$ such that each R_i is either in Σ or is a resolvent of a pair of clauses in $\{R_1, \ldots, R_{i-1}\}$. In effect, the Subsumption Theorem tells us that logical implication can be decomposed into a sequence of resolution steps, followed by a subsumption step:



Proof of Subsumption Theorem:

We will show that "only if" part of the theorem holds using the method of induction on the size of Σ:

1. Let $|\Sigma| = 1$. That is $\Sigma = \{\alpha_1\}$. Let $\Sigma \models D$. Since the only result of applying resolution to Σ is α_1, we need to show that $\alpha_1 \subseteq D$. Suppose $\alpha_1 \not\subseteq D$. Let L be a literal in α_1 that is not in D. Let I be an interpretation that assigns L to true and all literals in D to false. Clearly I is a model for α_1 but not a model for D, which is not possible since $\Sigma \models D$. Therefore, $\alpha_1 \subseteq D$.

2. Let the theorem hold for $|\Sigma| = n$. We will see that it follows that it holds for $|\Sigma| = n + 1$. Let $\Sigma = \{\alpha_1, \ldots, \alpha_{n+1}\}$ and $\Sigma \models D$. By the Deduction Theorem, we know that this means $\Sigma - \{\alpha_{n+1}\} \models (D \leftarrow \alpha_{n+1})$, or $\Sigma - \{\alpha_{n+1}\} \models (D \vee \neg\alpha_{n+1})$. Let us set $\Sigma' = \Sigma - \{\alpha_{n+1}\}$, and let $L_1, \ldots, L_k$ be the literals in α_{n+1} that do not appear in D. That is $\alpha_{n+1} = L_1 \vee \cdots \vee L_k \vee D'$, where $D' \subseteq D$. $\Sigma' \models (D \vee \neg\alpha_{n+1})$, you should be able to see that $\Sigma' \models (D \vee \neg L_i)$ for $1 \leq i \leq k$. Since $|\Sigma'| = n$ and we

believe, by the induction hypothesis, that the subsumption theorem holds for $|\Sigma'| = n$, there must be some β_i for each L_i, such that $\Sigma' \vdash_R \beta_i$ and $\beta_i \subseteq (D \vee \neg L_i)$. Suppose $\neg L_i \notin \beta_i$. Then $\beta_i \subseteq D$, which means that β_i subsumes D. Since $\Sigma' \models \beta_i$ and $\Sigma \models \Sigma'$, the result follows. Now suppose $\neg L_i \in \beta_i$. That is, $\beta_i = \neg L_i \vee \beta_i'$, where $\beta_i' \subseteq D$. Clearly, we can resolve this with $\alpha_{n+1} = L_1 \vee \cdots \vee L_i \vee \cdots \vee L_k \vee D'$ to give $L_1 \vee \cdots \vee L_{i-1} \vee \beta_i' \vee \cdots \vee L_k \vee D'$. Progressively resolving against each of the β_i, we will be left with the clause $C = \beta_1' \vee \beta_2' \vee \cdots \vee \beta_k' \vee D'$. Since C is the result of resolutions using a clause from Σ (that is α_{n+1}) and clauses derivable from $\Sigma' \subset \Sigma$, it is evident that $\Sigma \vdash_R C$. Also, since $\beta_i' \subseteq D$ and $D' \subseteq D$, $C \subseteq D$ and the result follows.

You should be able to see that the proof in the other direction (the "if" part) follows easily enough from the soundness of resolution. □

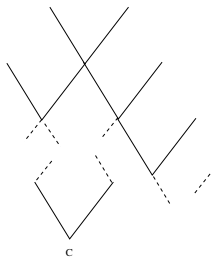
An immediate consequence of the Subsumption Theorem is that refutation-completeness of resolution follows.

Proof of Theorem 8

Recall what refutation completeness of resolution means: if Σ is a set of clauses that is unsatisfiable, then the empty clause □ is derivable using resolution. If Σ is unsatisfiable, then $\Sigma \models \square$. From the Subsumption Theorem, we know that if $\Sigma \models \square$, there must be a clause C such that $\Sigma \vdash_R C$ and C subsumes □. But the only clause subsuming □ is □ itself. Hence $C = \square$, which means that if $\Sigma \models \square$ then $\Sigma \vdash_R \square$.

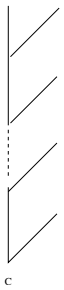
Proofs Using Resolution

So far, we have no strategy for directing the clauses obtained using resolution. Clauses are derived using any pair of clauses with complementary literals, and the process simply continues until we find the clause we want (for example, □ if we are interested in a proof of unsatisfiability). This procedure is clearly quite inefficient, since there is almost nothing constraining a proof, other than the presence of complementary literals. A "proof" for a clause C then ends up looking something like this:



Being creatures of limited patience and resources, we would like more directed approach. We can formalise this by changing our notion of a derivation. Recall

what we have been using so far: the derivation of a clause C from a set of clauses Σ means there is a sequence of clauses $R_1, \ldots, R_k = C$ such that each C_i is either in Σ or is a resolvent of a pair of clauses in $\{R_1, \ldots, R_{i-1}\}$. This results in the unconstrained form of a proof for C. We will say that there is a *linear* derivation for C from Σ if there is a sequence $R_0, \ldots, R_k = C$ such that $R_0 \in \Sigma$ and each R_i $(1 \leq i \leq k)$ is a resolvent of R_{i-1} and a clause $C_i \in \Sigma \cup \{R_0, \ldots, R_{i-2}\}$. With a little thought, you should be able to convince yourself that this will result in a derivation with a "linear" look:



Here, you can see that each new resolvent forms one of the clauses for the next resolution step. The other clause—sometimes called the "side clause"—can be any one of the clauses in Σ or a previous resolvent. For reasons evident from the diagram above, the proof strategy is called linear resolution, and we will extend our notation to indicate both the inference rule and the proof strategy. Thus $\Sigma \vdash_{LR} C$ will mean that C is derived from Σ using linear resolution. We can restrict things even further, by requiring side clauses to be only from Σ. The resulting proof strategy, called *input resolution* is important as it is a generalised form of SLD resolution, first mentioned on page 28.

While the restrictions imposed by the proof strategies ensure that proofs are more directed (and hence efficient), it is important at this point to ask: at what cost? Of course, since we are still using resolution as an inference rule, the individual (and overall) inference steps remain sound. But what about completeness? By this we mean refutation-completeness, since this is the only kind of completeness we were able to show with unconstrained resolution. In fact, it is the case that linear resolution retains the property of refutation-completeness, but input resolution for arbitrary clauses does not. That input resolution is not refutation complete can be proved using a simpe counter-example:

$$C_0 : \mathsf{Fred\ is\ an\ ape} \leftarrow \mathsf{Fred\ is\ human}$$
$$C_1 : \mathsf{Fred\ is\ an\ ape} \leftarrow \neg\ \mathsf{Fred\ is\ human}$$
$$C_2 : \neg\ \mathsf{Fred\ is\ an\ ape} \leftarrow \mathsf{Fred\ is\ human}$$
$$C_3 : \neg\ \mathsf{Fred\ is\ an\ ape} \leftarrow \neg\ \mathsf{Fred\ is\ human}$$

Now, a little effort should convince you that this set of clauses is unsatisfiable. But input resolution will simply yield a sequence of resolvents: Fred is human, Fred is an ape, Fred is human, ...

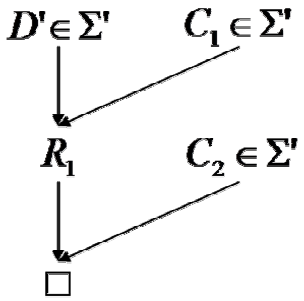


Figure 1.4: Part of case 1 of the proof for theorem 10.

Theorem 10 *If Σ is an unsatisfiable set of clauses, and $C \in \Sigma$ such that $\Sigma \backslash \{C\}$ is satisfiable, then there is a linear refutation of Σ with C as top clause.*

Proof:

We can assume Σ is finite. Let n be the number of distinct atoms occurring in literals in literals in Σ. We prove the lemma by induction on n

1. If $n = 0$, then $\Sigma = \{\Box\}$. Since $\Sigma \setminus \{C\}$ is satisfiable, $C = \Box$.

2. Suppose the lemma holds for $n \leq m$, and suppose $m + 1$ distinct atoms appear in Σ. We distinguish two cases.

   - *Case 1*: Suppose $C = L$, where L is a literal. We first delete all clauses from Σ which contain the literal L (so we also delete C itself from Σ). Then we replace clauses which contain the literal $\neg L$ by clauses constructed by deleting these $\neg L$ (so for example, $L_l \vee \neg L \vee L_2$ will be replaced by $L_1 \vee L_2$). Call the finite set obtained in this way Γ.

     Note that neither the literal L, nor its negation, appears in clauses in Γ. If M were a Herbrand model of Γ, then $M \cup \{L\}$ (*i.e.*, the Herbrand interpretation which makes L true, and is the same as M for other literals) would be a Herbrand model of Σ. Thus since Σ is unsatisfiable, Γ must be unsatisfiable.

     Now let Σ' be an unsatisfiable subset of Γ, such that every proper subset of Σ' is satisfiable. Σ' must contain a clause D' obtained from a member of Σ which contained $\neg L$, for otherwise the unsatisfiable set Σ' would be a subset of $\Sigma \backslash \{C\}$, contradicting the assumption that $\Sigma \setminus \{C\}$ is satisfiable. By construction of Σ', we have that $\Sigma' \setminus \{D'\}$ is satisfiable. Furthermore, Σ' contains at most m distinct atoms, so by the induction hypothesis there exists a linear refutation of Σ' with top clause D'. See the Figure 1.4 for illustration.

     Each side clause in this refutation that is not equal to a previous center clause, is either a member of Σ or is obtained from a member

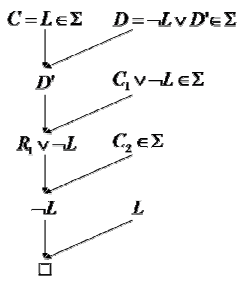


Figure 1.5: The complete picture of case 1 of the proof for theorem 10.

of Σ by means of the deletion of $\neg L$. In the latter kind of side clauses, put back the deleted $\neg L$ literals, and add these $\neg L$ to all later center clauses. Note that afterwards, these center clauses may contain multiple copies of $\neg L$. In particular, the last center clause changes from $\Box$ to $\neg L \vee \ldots \vee \neg L$. Since D' is a resolvent of C and $D = \neg L \vee D' \in \Sigma$, we can add C and D as parent clauses on top of the previous top clause D'. That way, we get a linear derivation of $\neg L \vee \ldots \vee \neg L$ from Σ, with top clause C. Finally, the literals in $\neg L \vee \ldots \vee \neg L$ can be resolved away using the top clause $C = L$ as side clause. This yields a linear refutation of Σ with top clause C (see Figure 1.5).

- *Case 2:* Exercise

□

The incompleteness of input also means that the Subsumption Theorem will not hold for input resolution in general. What, then, can we say about SLD resolution? The short answer is that it too is incomplete. But, for a restricted form of clauses, input and SLD resolution *are* complete. The restriction is to Horn clauses: recall that these are clauses that have at most 1 positive literal. Indeed, it is this restriction that forms the basis of theorem-proving in the PROLOG language, which is restricted (at least in its pure form) to Horn clauses, albeit in first-order logic (but the result still holds in that case as well).

Proofs Using SLD Resolution

Before we get to SLD, we first make our description of input resolution a little more precise: the derivation of a clause C from a set of clauses Σ using input resolution means there is a sequence of clauses $R_0, \ldots, R_k = C$ such that $R_0 \in \Sigma$ and each R_i $(1 \leq i \leq k)$ is a resolvent of R_{i-1} and a clause $C_i \in \Sigma$. Now, we add further restrictions. Let Σ be a set of Horn clauses. Further, let R_i be a resolvent of a selected negative literal in R_{i-1} and the positive literal of a definite clause $C_i \in \Sigma$. The selection rule is called the "computation rule" and

the resulting proof strategy is called SLD resolution ("Selected Linear Definite" resolution). We illustrate this with an example. Let Σ be the set of clauses:

C_0 : $\neg$ Fred is an ape

C_1 : Fred is an ape $\leftarrow$ Fred is human, Fred has hair

C_2 : Fred is human

C_3 : Fred has hair

A little thought should convince you that $\Sigma \models \square$. We want to see if $\Sigma \vdash_{SLD} \square$. It is evident that C_0 and C_1 resolve. The resolvent is R_1:

C_0 : $\neg$ Fred is an ape

C_1 : Fred is an ape $\leftarrow$ Fred is human, Fred has hair

R_1 : $\leftarrow$ Fred is human, Fred has hair

C_2 : Fred is human

C_3 : Fred has hair

Since we are using SLD, one of the resolvents for the next step has to be R_1. The other resolvent has to be one of the C_i's. Suppose our selection rule selects the "rightmost" literal first for resolution (that is, $\neg$ Fred has hair in R_1) This resolves with C_3, giving R_2 : $\neg$ Fred is human, which in turn resolves with C_2 to give $\square$. The SLD (input) resolution diagram for this is presented in Figure 1.6.

It is more common, especially in the logic-programming literature, to present instead the search process confronting a SLD-resolution theorem prover in the form of a tree-diagram, called an *SLD-tree*. Such a tree effectively contains all possible derivations that can be obtained using a particular literal selection rule. Each node in the tree is a "goal" of the form $\leftarrow L_1, L_2, \ldots, L_k$. That is, it is a clause of the form $(\neg L_1 \vee \neg L_2 \vee \cdots \vee \neg L_k)$. Given a set of clauses Σ, the children of a node in the SLD-tree are the result of resolving with clauses in Σ (nodes representing the empty clause $\square$ have no children). The SLD-tree for the example we just looked at is shown in Figure 1.7

We can now see what refutation-completeness for Horn clauses for SLD-resolution means in terms of SLD-trees. In effect, this means that if a set of clauses is unsatisfiable then there will be a leaf in the SLD-tree with the empty clause $\square$. Further, the computation rule will not alter this (informally, you can see that different computation rules will simply move the location of the $\square$ around). We will have more to say on SLD-resolution with first-order logic in a later section. There, we will see that in addition to the computation rule, we will also need a "search" rule that determines how the SLD-tree is searched. Search trees there can have infinite branches, and although completeness is unaffected by the choice of the computation rule (that is, there will be a $\square$ in the tree if the set of first-order clauses is unsatisfiable), we may not be able to reach it with a fixed search rule.

Theorem 11 *If Σ is an unsatisfiable set of horn clauses, then there is an SLD refutation of Σ.*

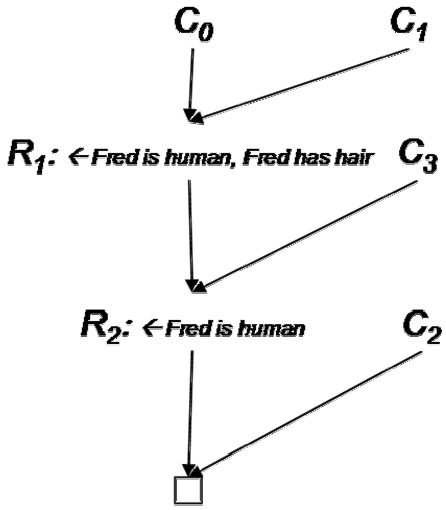


Figure 1.6: Example of SLD-deduction of $\square$ from Σ.

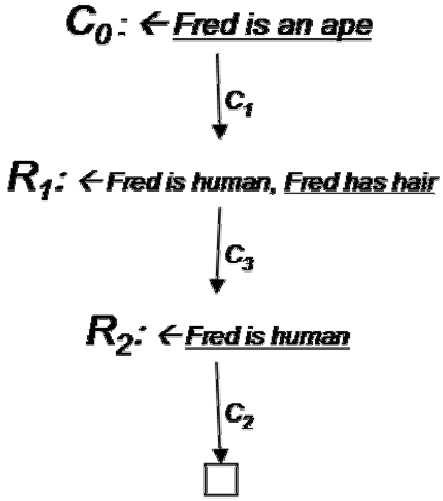


Figure 1.7: Example SLD-tree with C_0 at root.

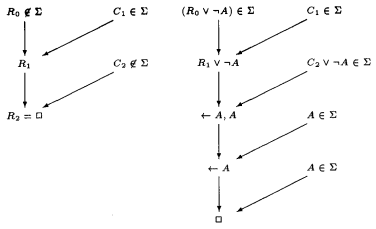


Figure 1.8: Illustration of the proof for theorem 11. The SLD refutation of Σ' is on the left and that for Σ is on the right.

Proof:

We can assume Σ is finite. Let n be the number of facts (clauses consisting of a single positive literal) in Σ. We prove the lemma by induction on n

1. If $n = 0$, then $\square \in \Sigma$ for otherwise the empty set would be a Herbrand model of I.

2. Suppose the lemma holds for $0 \leq n \leq m$, and suppose $m+1$ distinct facts appear in Σ. If $\square \in \Sigma$ the lemma is obvious, so suppose $\square \notin \Sigma$.

   Let, A be a fact in Σ. We first delete all clauses from Σ; which have A as head (so we also delete the fact A from Σ). Then we replace clauses which have A in their body by clauses constructed by deleting these atoms A from the body (so for example, $B \leftarrow A, B_1, \ldots, B_k$ will be replaced by $B \leftarrow B_1, \ldots, B_k$). Call the set obtained in this way Σ'. If M is a model of Σ', then $M \cup \{A\}$ is a Herbrand model of Σ. Thus since Σ is unsatisfiable, Σ' must be unsatisfiable. Σ' only contains m facts, so by the induction hypothesis, there is an SLD-refutation of Σ'. If this refutation only uses clauses from Σ' which were also in Σ then this is also an SLD-refutation of Σ, so then we are done. Otherwise, if C is the top clause or an input clause in this refutation and $C \notin \Sigma$, then C was obtained from some $C' \in \Sigma$ by deleting all atoms A from the body of Σ. For all such C, do the following: restore the previously deleted copies of A to the body of C (which turns C into C' again), and add these atoms A to all later resolvents. This way, we can turn the SLD-refutation of Σ' into an SLD-derivation of $\leftarrow A, \ldots, A$ from Σ. See Figure 1.8 for illustration, where we add previously deleted atoms A to the bodies of R_0 and C_2. Since also $A \in \Sigma$, we can construct an SLD-refutation of Σ, using A a number of times as input clause to resolve away all members of the goal $\leftarrow A, \ldots, A$.

□

1.3.5 Davis-Putnam Procedure

The inference problem addressed so far (particularly through the resolution procedure) is to determine if a proposition α logically follows from a given logical theory Σ. As we saw, this is achieved by reducing the problem to a coNP-complete[15] unsatisfiability problem; based on the contradiction theorem, it amounts to negating the goal formula α, add it to the theory and test the conjunction for unsatisfiability.

However, often, one is faced with the requirement for a model M for a logical theory Σ. This can turn out to be easier problem than the usual problem of inference, since it is enough to find one model for the theory, as against trying all possible truth assignments as in the case of solving an unsatisfiability problem. For example, theory might describe constraints on the different parts of a car. And you are interested in a model that satisifes all the constraints. In terms of search, you search the space of assignment and stop when you find an assignment that satisifies the theory.

While the resolution procedure can be modified so that it gives you a model, the Davis-Putnam-Logemann-Loveland (DPLL) procedure is a more efficient procedure for solving SAT problems. Given a set of clauses Σ defined over a set of variables $\mathcal{V}$, the Davis-Putnam procedure $DPLL(\Sigma)$ returns 'satisfiable' if is satisfiable. Otherwise return 'unsatisfiable'.

The $DPLL(\Sigma)$ procedure consists of the following steps. The first two steps specify termination conditions. The last two rules actually work on the clauses in Σ.

1. If $\Sigma = \emptyset$, return 'satisfiable'. This convention was introduced on pages 22 when we introduced logical entailment, as also in the footnote on page 29.

2. If $\square \in \Sigma$ return 'unsatisfiable'. This convention was discussed in the footnote on page 29.

3. **Unit-propagation Rule:** If Σ contains any unit-clause $C = \{c\}$ (*c.f.* page 29 for definition), assign a truth-value to the variable in literal c that satisfies c, 'simplify' Σ to Σ' and (recursively) return $DPLL(\Sigma')$. The rationale here is that if Σ has any unit clase $C = \{c\}$, the only way to satisfy C is to make c true. The simplification of Σ to Σ' is achieved by:

   (a) removing all clauses from Σ that contain the literal c (since all such clauses will now be true)

   (b) removing the negation of literal c (*i.e.*, $\neg c$) from every clause in Σ that contains it

[15] A decision problem C is Co-NP-complete if it is in Co-NP and if every problem in Co-NP is polynomial-time many-one reducible to it. The problem of determining whether a given boolean formula is tautology is a coNP-complete problem as well. A problem C is a member of co-NP if and only if its complement $\overline{C}$ is in complexity class NP. For example, the satisfiability problem is an NP-complete problem. Therefore the unsatisfiability problem is a coNP-complete problem.

4. **Splitting Rule:** Select from $\mathcal{V}$, a variable v which has not been assigned a truth-value. Assign one truth value t to it, simplify Σ to Σ' and (recursively) call $DPLL(\Sigma')$.

   (a) If the call returns 'satisfiable' (*i.e.*, we made a right choice for truth value of v), then return 'satisfiable'.

   (b) Otherwise (that is, if we made a wrong choice for truth value of v), assign the other truth-value to v in Σ, simplify to Σ'' and return $DPLL(\Sigma'')$.

The DPLL procedure can construct a model (if there exists one) by doing a book-keeping over all the assignments. This procedure is complete (that is, it constructs a model if there exists one), correct (the procedure always finds a trutn assignment that is a model) and guaranteed to terminate (since the space of possible assignments is finite and since DPLL explores that space systematically). In the worst case, DPLL requires exponential time, owing to the splitting rule. This is not surprissing, given the NP-completeness of the SAT problem. Heuristics are needed to determine (i) which variable should be instantiated next and (ii) to what value the instantiated varaible should be set. In all SAT competitions[16] so far, Davis Putnam-based procedures have shown the best performance.

As an illustration, we will use the DPLL procedure to determine a model for $\Sigma = \{\{a, b, \neg c\}, \{\neg a, \neg b\}, \{c\}, \{a, \neg b\}\}$. Since Σ is neither an empty set nor contains the empty clause, we move on to step 3 of the procedure. Σ contains a single unit clause $\{c\}$, which we will set to true and simplify the theory to $\Sigma^1 = \{\{a, b\}, \{\neg a, \neg b\}, \{a, \neg b\}\}$. Next, we apply the splitting rule. Let us choose a and set it to 'false'. This yields $\Sigma^2 = \{\{b\}, \{\neg b\}\}$. It can be verified that $\Sigma^2 \equiv \{\Box\}$. We therefore backtrack and set a to 'true'. This yields $\Sigma^3 = \{\{\neg b\}\}$. Thereafter, application of unit propagation yields $\Sigma^4 = \{\}$, which is satisfiable according to step 1 of the procedure. Thus, using the DPLL procedure, we obtain a model for Σ as $M = \{a, c\}$.

As another example, consider $\Sigma = \{\{a, \neg b, \neg c, \neg d\}, \{b, \neg d\}, \{c, \neg d\}, \{d\}\}$. As we will see, application to DPLL to this problem does not involve any backtracking, making the task relatively easier. Σ is neither an empty set nor contains the empty clause. Hence we apply the unit propagation rule 3 on $\{d\}$ to obtain $\Sigma^1 = \{\{a, \neg b, \neg c\}, \{b\}, \{c\}\}$. We can again apply unit propagation to Σ^1 on $\{b\}$ to obtain $\Sigma^2 = \{\{a, \neg c\}, \{c\}\}$. Finally, we can apply unit propagation to Σ^2 on $\{c\}$ and then to $\{a\}$ obtain $\Sigma^4 = \{\}$, which is satisfiable. This yields a model $M = \{d, b, c, a\}$ of Σ.

The DPLL procedure is similar to the traditional constraint propagation procedure. The splitting rule in DPLL is similar to the backtracking step in constraint propagation, while the unit propagation rule is similar to the unavoidable steps of consistency checking and forward propagation in traditional CP.

[16] `http://www.satcompetition.org/`

Phase Transitions

We saw that in the worst case, DPLL requires exponential time. Couldn't we do better in the average case? For CNF-formulæ in which the probability for a positive appearance, negative appearance and non-appearance in a clause is 1/3, DP needs on average quadratic time [Gol79]. In retrospect, it was discovered that the formulæ in [Gol79] have a very high probability of being satisfiable. Thus, these formulæ are not representative of those encountered in practice. The idea of *phase transition* was conjectured by [CKT91] to identify hard to solve problem instances:

> *All NP-complete problems have at least one order parameter and the hard to solve problems are around a critical value of this order parameter. This critical value (a phase transition) separates one region from another, such as over-constrained and under-constrained regions of the problem space.*

This conjecture was initially confirmed for the graph coloring and Hamilton path problems and later for other NP-complete problems, including SAT. In the case of SAT problem, an example of the order parameter is ratio of the number of variables to the number of clauses.

1. For higher settings of the parameter, the problem is over-constrained (the formulæ are unsatisfiable). If the probability of a solution is close to 0, this fact can usually be determined early in the search.
2. For lower settings of the parameter, the problem is under-constrained (the formulæ are easily satisfiable). When the probability of a solution is close to 1, there are many solutions, and the first search path of a backtracking search is usually successful.

When this parameter is varied, the problem moves from the over-constrained to the under-constrained region (or vice versa). At phase the transition points, half of the problems are satisfiable and half are not. It is typically in this region that algorithms have difficulty in solving the problem[17]. Cook and Mitchell [CM97] empirically found that for a 3-SAT problem, the phase transition occurs at a clause:variables ratio of around 4.3 (in this experiment, clauses were generated by choosing variables for a clause and complementing each variable with probability 0.5). As an illustration, in the 2003 version of the SAT competition, the largest instances solved using greedy SAT solvers consisted of $100,000$ variables and $1,000,000$ clauses (clause:variable ratio of 10), whereas the smallest unsolved instances comprised 200 variables and $1,000$ clauses (clause:variable ratio of 5). It was also reported in [CM97] that the runtime for the DPLL procedure peaks at the phase transition. In the phase transition region, the DPLL algorithm often near successes. Many benchmark problems are located in the phase transition region, though they have a special structure in addition.

[17] Note that hard instances can also exist in regions of the more easily satisfiable or unsatisfiable instances.

1.3.6 Local Search Methods

Local search methods are standard search procedures for optimization problems. A local search method explores the neighborhood of the current solution and tries to enhance the solution till it cannot do any better. The hope is to produce better configurations through local modifications. The value of a configuration in a logical problem could be measured using the number of satisfied constraints/clauses. However, for logical problems, local maxima are inappropriate; it is required to satisfy all clauses in the theory and not just some. But through random restarts or by noise injection, local maxima can be escaped. In practice, local search performs quite well for finding satisfying assignments of CNF formulæ, especially for under-constrained or over-constrained SAT problems.

GSAT and WalkSat [SKC93] are two local search algorithms to solve boolean satisfiability problems in CNF. They start by assigning a random value to each variable. If the assignment satisfies all clauses, the algorithm terminates, returning the assignment. Otherwise, an unsatified clause is selected and the value of exactly one variable changed. Due to the conjunctive normal form, flipping one variable will result in that clause becoming satisfied. The above is then repeated until all the clauses are satisfied. WalkSAT and GSAT differ in the methods used to select the variable to flip. While GSAT makes the change which minimizes the number of unsatisfied clauses in the new assignment, WalkSAT selects the variable that, when flipped, results in no previously satified clauses becoming unsatified (some sort of downward compatibility requirement). MaxWalkSat is a variant of WalkSat designed to solve the weighted satisfiability problem, in which each clause is associated with a weight. The goal in MaxWalkSat is to find an assignment (which may or may not satisfy the entire formula) that maximizes the total weight of the clauses satisfied by that assignment. These algorithms perform very well on the randomly generated formulæ in the phase transition region. Monitoring the search procedure of these greedy solvers reveals that in the begginning, each procedure is very good at reducing the number of unsatisfied clauses. However, it takes a long time to satisfy the few remaining clauses (called plateaus). The GSAT algorithm is outlined in Figure 1.9.

1.3.7 Default inference under closed world assumption

Given any set of formulae Σ, the closed-world assumption is the assumption that Σ determines all the knowledge there is to be had about the formulae in the language. Thus, if we consider any proposition[18] A then A is taken to be true exactly when Σ (logically) implies A, but is otherwise taken to be false.

The closed-world assumption underlies the mode of reasoning known as *default inference*. There are many situations, both in ordinary daily life and in specific technical computing matters (such as explaining the theory of finite failure and the relationship it bears to reasoning with negative information), when

[18] In the first order logic programming context, the propositions in which we are primarily interested are the atoms of the Herbrand base.

INPUT: A set of clauses Σ, MAX-FLIPS, and MAX-TRIES.
OUTPUT: A satisfying truth assignment of Σ, if found.
for $i = 1$ to MAX-TRIES **do**
  T = a randomly-generated truth assignment.
  for j:=1 to MAX-FLIPS **do**
    if T satisfies Σ **then**
      return T
    end if
    v = a propositional variable such that a change in its truth assignment gives the largest increase in the number of clauses of Σ that are satisfied by T.
    $T = T$ with the truth assignment of v reversed.
  end for
end for
return "Unsatisfiable".

Figure 1.9: Procedure GSAT.

default inference is a necessary supplement to deductive inference. Consider this simple example of a single clause axiom $\Sigma = \{A \leftarrow B\}$ for which $B(\Sigma)$ is just $\{A, B\}$. Under the closed-world assumption, we may infer $\neg A$ for any ground atom $A \in B(\Sigma)$ that us not implied by Σ. This is a constrained[19] application of the rule of default inference:

Infer $\neg A$ in default of Σ implying A

Motivated by the desire to draw sound conclusions about negative information, we will consider two constructions that provide consequence-oriented meaning for default inference under the closed-world assumption.

1. CWA(Σ) : The combination of Σ with the default conclusions inferred from it is denoted by CWA(Σ) and is defined by

$$\text{CWA}(\Sigma) = \Sigma \cup \{\neg A \mid A \in B(\Sigma) \textit{ and not } \Sigma \models A\}$$

For the above example, $\text{CWA}(\Sigma) = \{A \leftarrow B, \neg A, \neg B\}$.

Soundness for the default inference of negative conclusions under the closed-world assumption can be referred to the the logical construction CWA(Σ) as follows:

$$for\ all\ A \in B(\Sigma),\ \text{CWA}(\Sigma) \models \neg A\ if\ \Sigma \vdash_{CWA} \neg A$$

There some practical problems with CWA:

[19] Constrained because (i) Σ is assumed to be inconsistent, else Σ would necessarily imply both A and $\neg A$ (ii) only the case where A is atomic is considered, otherwise if Σ implied, say, neither A nor $\neg A$, then the default rule would infer both $\neg A$ and $\neg A$, which would again be inconsistent.